Javascript

▼ Topics

I. Fundamentals (Building Blocks)

- **Data Types:** Numbers, Strings, Booleans, Null, Undefined, Symbols, BigInts, Objects
- **Variables:** `var`, `let`, `const` (scope, hoisting)
- **Operators:** Arithmetic, Assignment, Comparison (== vs. ===), Logical, Ternary
- **Control Flow:** `if/else`, `switch`, `for`, `while`, `do...while` loops
- **Functions:** Declaration, Expression, Arrow functions, Parameters, Return values
- **Scope:** Global, Function, Block scope
- **Hoisting:** Variable and Function hoisting
- **Type Coercion:** Understanding how JavaScript converts types automatically

II. Intermediate (Core Concepts)

- **Objects and Prototypes:** Object literals, Constructors, `this` keyword, Prototypes, Inheritance (prototype chain)
- **Arrays:** Array methods (`push`, `pop`, `shift`, `unshift`, `map`, `filter`, `reduce`, `forEach`, etc.)
- **DOM Manipulation (Browser Environment):** Selecting elements (e.g., `getElementById`, `querySelector`), Modifying content, Attributes, Styling, Event listeners
- **Events:** Event types (click, submit, keypress, etc.), Event handling, Event propagation (bubbling, capturing)
- **Asynchronous JavaScript:** Callbacks, Promises (`.then`, `.catch`, `.finally`), Async/Await
- **Closures:** Understanding how inner functions access outer function variables
- `this` **Keyword:** Context-dependent `this` (global, function, method, arrow functions)

III. Advanced (Deepening Knowledge)

- **Modules:** Import/Export (ES Modules), CommonJS (Node.js)
- **Error Handling:** `try...catch` blocks
- **Regular Expressions:** Pattern matching, Searching, Replacing
- **Design Patterns:** Common solutions to recurring problems (e.g., Singleton, Factory, Observer)
- **Testing:** Unit testing, Integration testing (using frameworks like Jest, Mocha)
- **Performance Optimization:** Techniques for writing efficient JavaScript code
- **Security Best Practices:** Preventing common vulnerabilities (e.g., XSS, CSRF)
- **Working with APIs:** Fetching data from external sources (REST APIs, GraphQL)
- **Web Sockets:** Real-time communication between client and server

IV. Beyond (Specialized Areas)

- **Frameworks and Libraries:** React, Angular, Vue.js (front-end), Node.js, Express.js (back-end)
- **Version Control (Git):** Managing code changes
- **Build Tools:** Webpack, Babel (bundling, transpiling)
- **Deployment:** Deploying web applications

- **Server-Side JavaScript (Node.js):** Building back-end applications
- **Databases:** Working with databases (SQL, NoSQL)
- **TypeScript:** Adding static typing to JavaScript
- **WebAssembly:** Running code written in other languages in the browser

I. Fundamentals (Building Blocks)

▼ **Data Types:** Numbers, Strings, Booleans, Null, Undefined, Symbols, BigInts, Objects

1. Primitive Data Types (Immutable)

These data types hold a single value and are immutable, meaning their values cannot be changed directly.

Number: Represents numeric values, including integers and floating-point numbers.
JavaScript

```
let age = 30;         // Integer
let price = 99.99;    // Floating-point
let pi = 3.14159;     // Floating-point
let infinity = Infinity; // Special numeric value
let nan = NaN;        // Not a Number (also a special numeric value)

console.log(typeof age);      // "number"
console.log(typeof price);    // "number"
console.log(typeof infinity); // "number"
console.log(typeof nan);      // "number" (a quirk!)
```

String: Represents textual data, enclosed in single quotes (''), double quotes (""), or backticks (``). Backticks allow f
or template literals (string interpolation).
JavaScript

```
let name = "John Doe";
let message = 'Hello, world!';
let greeting = `Hi, ${name}!`; // Template literal

console.log(typeof name);     // "string"
console.log(typeof message);  // "string"
console.log(typeof greeting); // "string"
```

Boolean: Represents a logical value: true or false.
JavaScript

```
let isLoggedIn = true;
let isAdult = false;

console.log(typeof isLoggedIn); // "boolean"
console.log(typeof isAdult);    // "boolean"
```

Null: Represents the intentional absence of a value. It's different from undefined.
JavaScript

```
let userAddress = null;  // Explicitly setting the variable to null

console.log(typeof userAddress); // "object" (a historical quirk)
```

Undefined: Represents a variable that has been declared but not assigned a value.
JavaScript

```
let cityName; // Declared but not assigned

console.log(typeof cityName); // "undefined"
```

Symbol (ES6): Represents a unique and immutable value, often used as keys for object properties to avoid naming collisions.

JavaScript

```
const uniqueId = Symbol('id');
const anotherUniqueId = Symbol('id'); // Different from uniqueId

console.log(uniqueId === anotherUniqueId); // false (Symbols are unique)
console.log(typeof uniqueId); // "symbol"
```

BigInt (ES2020): Represents integers of arbitrary precision, useful for numbers larger than the Number type can handle accurately. Created by appending n to a number literal.

JavaScript

```
const largeNumber = 1234567890123456789012345678901234567890n;

console.log(typeof largeNumber); // "bigint"
```

2. Non-Primitive Data Type (Mutable)

Object: Represents a collection of key-value pairs (properties). Objects are mutable, meaning their properties can be changed after creation. Arrays are a special type of object.

JavaScript

```
const person = {
  name: "Jane Doe",
  age: 25,
  city: "New York"
};

person.age = 26; // Modifying a property (objects are mutable)

const colors = ["red", "green", "blue"]; // Array (also an object)
colors.push("yellow"); // Modifying the array

console.log(typeof person); // "object"
console.log(typeof colors); // "object"
console.log(Array.isArray(colors)) // true (To check if it is array)
```

Key Differences: Primitive vs. Non-Primitive

Immutability: Primitives are immutable; non-primitives are mutable.

Storage: Primitives are stored by value; non-primitives are stored by reference. This means when you work with objects, you're often working with a reference to the object in memory, not a copy of the object itself.

Example demonstrating Pass by Reference:

JavaScript

```
let myObject = { value: 10 };
let anotherObject = myObject; // Assigning the reference

anotherObject.value = 20;
```

```
console.log(myObject.value);    // 20 (myObject is also changed!)
console.log(anotherObject.value); // 20
```

▼ **Variables:** `var`, `let`, `const` (scope, hoisting)

1. Variable Declarations

JavaScript offers three ways to declare variables: var, let, and const. They differ primarily in how they handle scope and hoisting.

var: Historically the primary way to declare variables.

let (ES6): Introduced in ES6 (ECMAScript 2015), providing block scope.

const (ES6): Also introduced in ES6, used for declaring constants (values that should not be reassigned). Also block-scoped.

2. Scope

Scope determines the visibility and accessibility of variables within your code.

Function Scope (var): Variables declared with var have function scope. They are accessible within the entire function where they are declared, even inside nested blocks (like if statements or loops).
JavaScript

```
function myFunction() {
  var functionVar = "I'm function-scoped";
  if (true) {
    var blockVar = "I'm also function-scoped (due to var)";
  }
  console.log(functionVar); // "I'm function-scoped"
  console.log(blockVar);   // "I'm also function-scoped (due to var)"
}

myFunction();
// console.log(functionVar); // Error: functionVar is not defined outside myFunction
// console.log(blockVar);   // Error: blockVar is not defined outside myFunction
```

Block Scope (let, const): Variables declared with let and const have block scope. They are only accessible within the block (curly braces {}) where they are defined.
JavaScript

```
function myOtherFunction() {
  let blockLet = "I'm block-scoped";
  const blockConst = "I'm also block-scoped";
  if (true) {
    let innerBlockLet = "I'm from an inner block";
    const innerBlockConst = "I'm also from an inner block";
    console.log(blockLet);      // "I'm block-scoped"
    console.log(blockConst);    // "I'm also block-scoped"
    console.log(innerBlockLet); // "I'm from an inner block"
    console.log(innerBlockConst); // "I'm also from an inner block"
```

```
  }
  console.log(blockLet);     // "I'm block-scoped"
  console.log(blockConst);    // "I'm also block-scoped"
  // console.log(innerBlockLet); // Error: innerBlockLet is not defined outside the if block
  // console.log(innerBlockConst); // Error: innerBlockConst is not defined outside the if block
}

myOtherFunction();
// console.log(blockLet); // Error: blockLet is not defined outside myOtherFunction
// console.log(blockConst);// Error: blockConst is not defined outside myOtherFunction
```

Global Scope: Variables declared outside any function or block have global scope. They are accessible from anywh
ere in your code. It's generally best to minimize the use of global variables.
JavaScript

```
var globalVar = "I'm global"; // Using var (less recommended)
let globalLet = "I'm also global"; // Using let (more common)
const globalConst = "I'm a global constant"; // Using const

function myGlobalFunction() {
  console.log(globalVar);   // "I'm global"
  console.log(globalLet);   // "I'm also global"
  console.log(globalConst); // "I'm a global constant"
}

myGlobalFunction();
console.log(globalVar);   // "I'm global"
console.log(globalLet);   // "I'm also global"
console.log(globalConst); // "I'm a global constant"
```

3. Hoisting

Hoisting is JavaScript's behavior of moving variable declarations to the top of their scope during the compilation ph
ase. However, only the declarations are hoisted, not the initializations.

var Hoisting: Variables declared with var are hoisted to the top of their function scope and initialized to undefined.
JavaScript

```
console.log(hoistedVar); // undefined (declaration is hoisted, but not the assignment)
var hoistedVar = "I'm hoisted";
console.log(hoistedVar); // "I'm hoisted"

function myHoistingFunction() {
  console.log(innerHoistedVar); // undefined
  var innerHoistedVar = "I'm hoisted inside";
  console.log(innerHoistedVar); // "I'm hoisted inside"
}

myHoistingFunction();
```

let and const Hoisting: Variables declared with let and const are also hoisted, but they are not initialized. Trying to a
ccess them before their declaration1 results in a ReferenceError. This is often referred to as the "temporal dead zon
e" (TDZ).
1.
github.com
github.com

JavaScript

```
// console.log(letHoisted); // ReferenceError: Cannot access 'letHoisted' before initialization
let letHoisted = "I'm let hoisted";
console.log(letHoisted); // "I'm let hoisted"

// console.log(constHoisted); // ReferenceError: Cannot access 'constHoisted' before initialization
const constHoisted = "I'm const hoisted";
console.log(constHoisted); // "I'm const hoisted"
```

4. const Reassignment

const prevents reassignment of the variable itself, not necessarily the contents of the variable if it's an object or arr ay.

JavaScript

```
const myConstObject = { name: "Alice" };
myConstObject.name = "Bob"; // Allowed: Modifying a property of the object
console.log(myConstObject); // { name: "Bob" }

// myConstObject = { name: "Charlie" }; // Error: Assignment to constant variable

const myArray = [1, 2, 3];
myArray.push(4); // Allowed: Modifying the array
console.log(myArray); // [1, 2, 3, 4]

// myArray = [5, 6, 7]; // Error: Assignment to constant variable
```

Key Recommendations:

Use let and const for block scope whenever possible. This helps prevent accidental variable overwrites and makes your code more predictable.
Use const for values that should not be reassigned. This helps with code clarity and prevents unintended changes.
Avoid using var unless you have a specific reason to use function scope.
Be mindful of hoisting, especially with var. It's often best to declare variables at the top of their scope to avoid unex pected behavior.

▼ **Operators:** Arithmetic, Assignment, Comparison (== vs. ===), Logical, Ternary

1. Arithmetic Operators

These operators perform mathematical calculations.

Addition (+): Adds two operands.
Subtraction (-): Subtracts the second operand from the first.
Multiplication (*): Multiplies two operands.
Division (/): Divides the first operand by the second.
Modulo (%): Returns the remainder of a division.
Increment (++): Increases the operand by 1.
Decrement (--): Decreases the operand by 1.
Exponentiation (): Raises the first operand to the power of the second.

JavaScript

```
let x = 10;
let y = 5;
```

```
console.log(x + y);  // 15 (Addition)
console.log(x - y);  // 5 (Subtraction)
console.log(x * y);  // 50 (Multiplication)
console.log(x / y);  // 2 (Division)
console.log(x % y);  // 0 (Modulo)

let z = 3;
z++;
console.log(z);     // 4 (Increment)
z--;
console.log(z);     // 3 (Decrement)

console.log(2 ** 3); // 8 (Exponentiation)
```

2. Assignment Operators

These operators assign values to variables.

Assignment (=): Assigns the value on the right to the variable on the left.
Addition Assignment (+=): Adds the right operand to the left and assigns the result to the left.
Subtraction Assignment (-=): Subtracts the right operand from the left and assigns the result to the left.
*Multiplication Assignment (=): Multiplies the left operand by the right and assigns the result to the left.
Division Assignment (/=): Divides the left operand by the right and assigns the result to the left.
Modulo Assignment (%=): Calculates the modulo and assigns the result to the left.
JavaScript

```
let a = 10;
a += 5; // Equivalent to a = a + 5;
console.log(a); // 15

let b = 20;
b -= 3; // Equivalent to b = b - 3;
console.log(b); // 17

let c = 4;
c *= 2; // Equivalent to c = c * 2;
console.log(c); // 8

let d = 10;
d /= 2; // Equivalent to d = d / 2;
console.log(d); // 5

let e = 11;
e %= 3; // Equivalent to e = e % 3;
console.log(e); // 2
```

3. Comparison Operators

These operators compare two operands and return a Boolean value (true or false).

Equal To (==): Checks if two operands are equal (performs type coercion).
Not Equal To (!=): Checks if two operands are not equal (performs type coercion).
Strict Equal To (===): Checks if two operands are equal and of the same type (no type coercion).
Strict Not Equal To (!==): Checks if two operands are not equal or not of the same type.
Greater Than (>): Checks if the left operand is greater than the right.

Less Than (<): Checks if the left operand is less than the right.
Greater Than or Equal To (>=): Checks if the left operand is greater than or equal to the right.
Less Than or Equal To (<=): Checks if the left operand is less than or equal to the right.
JavaScript

```
console.log(5 == "5");   // true (type coercion)
console.log(5 === "5");  // false (strict equality, different types)
console.log(5 != "5");   // false (type coercion)
console.log(5 !== "5");  // true (strict inequality, different types)
console.log(10 > 5);     // true
console.log(10 < 5);     // false
console.log(10 >= 10);   // true
console.log(10 <= 5);    // false
```

Important: == vs. ===

== (loose equality) performs type coercion, meaning it tries to convert the operands to the same type before comp arison. This can lead to unexpected results.
=== (strict equality) does not perform type coercion. It only returns true if the operands are equal and of the same t ype. It is highly recommended to use === and !== whenever possible to avoid unexpected behavior.
4. Logical Operators

These operators combine or modify Boolean values.

Logical AND (&&): Returns true if both operands are true.
Logical OR (||): Returns true if at least one operand is true.
Logical NOT (!): Reverses the Boolean value of its operand.
JavaScript

```
let p = true;
let q = false;

console.log(p && q); // false (Logical AND)
console.log(p || q); // true (Logical OR)
console.log(!p);     // false (Logical NOT)
```

5. Ternary Operator (Conditional Operator)

This operator provides a concise way to write if...else statements.

Syntax: condition ? expressionIfTrue : expressionIfFalse
JavaScript

```
let age = 20;
let message = age >= 18 ? "Adult" : "Minor";
console.log(message); // "Adult"

let isRaining = true;
let activity = isRaining ? "Stay inside" : "Go outside";
console.log(activity); // "Stay inside"
```

▼ **Control Flow:** `if/else` , `switch` , `for` , `while` , `do...while` loops

1. if/else Statements

if/else statements execute different blocks of code depending on whether a condition is true or false.

if: Executes a block of code if a condition is true.
else if: (Optional) Executes a block of code if the previous if or else if conditions are false and the current condition is true.
else: (Optional) Executes a block of code if all previous if and else if conditions are false.
JavaScript

```
let age = 20;

if (age >= 18) {
  console.log("You are an adult.");
} else {
  console.log("You are a minor.");
}

let score = 85;

if (score >= 90) {
  console.log("A");
} else if (score >= 80) {
  console.log("B");
} else if (score >= 70) {
  console.log("C");
} else {
  console.log("D or F");
}
```

2. switch Statement

The switch statement evaluates an expression and matches its value against multiple case labels. It provides a more efficient alternative to multiple if/else if statements when comparing a single value against many possible values.

JavaScript

```
let day = "Monday";

switch (day) {
  case "Monday":
    console.log("It's Monday, time to work!");
    break;
  case "Tuesday":
  case "Wednesday":
  case "Thursday":
    console.log("It's a weekday.");
    break;
  case "Friday":
    console.log("Almost weekend!");
    break;
  case "Saturday":
  case "Sunday":
    console.log("It's the weekend!");
    break;
  default:
    console.log("Invalid day.");
```

```
}
```
break: The break keyword is crucial. It terminates the switch statement, preventing "fall-through" to the next case. I f you omit break, execution will continue to the next case label, even if its value doesn't match.
default: The default case is executed if none of the case labels match the expression's value.
3. for Loop

The for loop is used to iterate over a block of code a specific number of times.

JavaScript

```
for (let i = 0; i < 5; i++) {
  console.log("Iteration:", i);
}

const numbers = [10, 20, 30, 40, 50];

for (let i = 0; i < numbers.length; i++) {
  console.log("Element:", numbers[i]);
}

for (const number of numbers) { //for...of loop
  console.log("for of loop number: ", number);
}

for (const index in numbers) { //for...in loop, returns index of an array.
  console.log("for in loop index: ", index);
}
```
Initialization: let i = 0; (Executed once before the loop starts)
Condition: i < 5; (Checked before each iteration; the loop continues if the condition is true)
Increment/Decrement: i++; (Executed after each iteration)
4. while Loop

The while loop repeatedly executes a block of code as long as a condition is true.

JavaScript

```
let count = 0;

while (count < 3) {
  console.log("Count:", count);
  count++;
}
```
The condition is checked before each iteration. If the condition is initially false, the loop will not execute.
5. do...while Loop

The do...while loop is similar to the while loop, but it guarantees that the block of code will execute at least once.

JavaScript

```
let num = 5;

do {
  console.log("Num:", num);
  num--;
```

```
} while (num > 0);
```
The condition is checked after each iteration. Therefore, the block of code always executes at least once, even if th e condition is initially false.
Choosing the Right Loop

Use for loops when you know the number of iterations in advance.
Use while loops when you want to repeat a block of code as long as a condition is true, and the condition might be f alse initially.
Use do...while loops when you need to execute a block of code at least once, regardless of the initial condition.
Use for...of loops when you want to iterate over the values of an iterable object (like an array).
Use for...in loops when you want to iterate over the keys of an object. (or indexes of an array, though for...of is usual ly better for arrays).

▼ **Functions:** Declaration, Expression, Arrow functions, Parameters, Return values

1. Function Declaration

A function declaration defines a named function using the function keyword.

JavaScript

```
function greet(name) {
  return "Hello, " + name + "!";
}

let greetingMessage = greet("Alice");
console.log(greetingMessage); // "Hello, Alice!"
```
function greet(name): Declares a function named greet that takes a parameter name.
return "Hello, " + name + "!";: The return statement specifies the value the function will return. If a function doesn't have a return statement, it implicitly returns undefined.

2. Function Expression

A function expression defines a function as part of a larger expression, often assigned to a variable.

JavaScript

```
const multiply = function(a, b) {
  return a * b;
};

let product = multiply(5, 3);
console.log(product); // 15
```
The function is anonymous (it doesn't have a name), but it's assigned to the multiply variable.
Function expressions are not hoisted like function declarations.

3. Arrow Functions (ES6)

Arrow functions provide a concise syntax for writing function expressions.

JavaScript

```
const add = (x, y) => {
```

```
  return x + y;
};

let sum = add(10, 7);
console.log(sum); // 17

// Implicit return (for single-expression functions)
const square = num => num * num;

let squaredValue = square(4);
console.log(squaredValue); // 16

// No parameters
const sayHi = () => "Hi!";
console.log(sayHi()); // Hi!

const double = numbers => numbers.map(number => number * 2);
console.log(double([1,2,3])); // [2,4,6]
```

(x, y) => { ... }: Defines an arrow function with parameters x and y.
Implicit return: If the function body contains only a single expression, you can omit the curly braces and the return k eyword.
Arrow functions do not have their own this context (they inherit this from the surrounding scope).

4. Parameters

Parameters are variables that receive values (arguments) when a function is called.

Formal Parameters: The parameters defined in the function's definition.
Actual Arguments: The values passed to the function when it's called.
JavaScript

```
function power(base, exponent) {
  return Math.pow(base, exponent);
}

let result = power(2, 3); // 2 and 3 are actual arguments
console.log(result); // 8
```

Default Parameters (ES6): You can provide default values for parameters.
JavaScript

```
function greetUser(name = "Guest") {
  return "Hello, " + name + "!";
}

console.log(greetUser("John")); // "Hello, John!"
console.log(greetUser());       // "Hello, Guest!"
```

Rest Parameters (ES6): The rest parameter (...) allows a function to accept an indefinite number of arguments as an array.
JavaScript

```
function sumAll(...numbers) {
  let total = 0;
  for (let num of numbers) {
    total += num;
```

```
  }
  return total;
}

console.log(sumAll(1, 2, 3, 4, 5)); // 15
5. Return Values

The return statement specifies the value a function returns.

If a function doesn't have a return statement, it implicitly returns undefined.
A function can return any data type, including objects, arrays, and other functions.
JavaScript

function createPerson(name, age) {
  return {
    name: name,
    age: age
  };
}

let person = createPerson("Alice", 30);
console.log(person); // { name: "Alice", age: 30 }
Key Concepts:

Functions are reusable blocks of code.
Parameters allow functions to accept input.
Return values allow functions to produce output.
Arrow functions offer a concise syntax for function expressions.
Functions are first class citizens, meaning they can be assigned to variables, passed as arguments, and returned fr
om other functions.1
```

Best Practices for JavaScript Functions

1. **Descriptive Names:** Use clear, descriptive names that indicate the function's purpose.
2. **Single Responsibility:** Each function should ideally have one specific task.
3. **Pure Functions (where possible):** Pure functions produce the same output for the same input and have no side effects (modifying external variables or performing I/O). This improves testability and predictability.
4. **Avoid Global Variables:** Minimize the use of global variables within functions. Rely on parameters and return values.
5. **Use `const` for Function Expressions:** If you're using function expressions, declare them with `const` to prevent accidental reassignment.
6. **Use Arrow Functions for Concise Syntax:** Arrow functions are great for short, one-line functions, especially when used with array methods.
7. **Default Parameters:** Use default parameters to handle optional arguments gracefully.
8. **Rest Parameters for Variable Arguments:** Use rest parameters (`...`) when a function needs to accept a variable number of arguments.
9. **Clear Return Values:** Always explicitly return a value, even if it's `null` or `undefined`, to make the function's behavior clear.
10. **Document Your Functions:** Add comments or use JSDoc to explain the purpose, parameters, and return value of each function.

```javascript
// 1. Descriptive Name and Single Responsibility
/**
 * Calculates the total price of items in a shopping cart.
 *
 * @param {Array<{price: number, quantity: number}>} cartItems An array of cart items.
 * @returns {number} The total price of the items.
 */
const calculateTotalPrice = (cartItems) => {
  if (!Array.isArray(cartItems)) {
    return 0; // Handle invalid input
  }

  let total = 0;
  for (const item of cartItems) {
    if (typeof item.price === 'number' && typeof item.quantity === 'number') {
      total += item.price * item.quantity;
    }
  }
  return total;
};

// Example cart
const shoppingCart = [
  { price: 10, quantity: 2 },
  { price: 25, quantity: 1 },
  { price: 5, quantity: 3 },
];

console.log('Total Price:', calculateTotalPrice(shoppingCart)); // Output: Total Price: 60

// 2. Pure Function (Example)
const doubleNumber = (num) => num * 2;

console.log(doubleNumber(5)); // 10 (always the same output for the same input)

// 3. Arrow Function and Default Parameters
const greetUser = (name = 'Guest') => `Hello, ${name}!`;

console.log(greetUser('John')); // Hello, John!
console.log(greetUser()); // Hello, Guest!

// 4. Rest Parameters
const sumNumbers = (...numbers) => {
  let sum = 0;
  for (const num of numbers) {
    if(typeof num === 'number'){
      sum += num;
    }
  }
  return sum;
};

console.log(sumNumbers(1, 2, 3, 4, 5)); // 15
```

```
console.log(sumNumbers(1, 2, "3", 4, 5)); // 12 (ignores the string)

// 5. Function returning an object.
const createPerson = (name, age) => {
  if (typeof name !== "string" || typeof age !== "number"){
    return null; // or throw an error.
  }
  return {
    name, // shorthand for name: name,
    age, // shorthand for age: age,
  };
}

console.log(createPerson("John", 30)); //{name: 'John', age: 30}
console.log(createPerson(1, "abc")); //null
```

Key Improvements in the Examples

- **Input Validation:** The `calculateTotalPrice` and `createPerson` functions include checks to ensure that the input parameters are of the expected types. This makes the functions more robust and less prone to errors.
- **Error Handling:** The functions handle invalid input by returning a default value (`0` or `null`). In more complex scenarios, you might throw an error instead.
- **JSDoc Comments:** The `calculateTotalPrice` function is documented using JSDoc comments, which make it easier to understand the function's purpose, parameters, and return value.
- **Type Checking:** The functions make sure the data passed to them is the correct type.
- **Shorthand object properties:** Used the shorthand notation when creating the returned object.

▼ **Scope:** Global, Function, Block scope

1. Global Scope

- **Explanation:** Variables declared outside any function or block reside in the global scope. They are accessible from anywhere in your JavaScript code.
- **Use Cases:**
  - Configuration settings that need to be accessed throughout the application.
  - Libraries or utility functions that need to be globally available.
  - In very simple scripts where scope management is less critical.
- **Best Practices:**
  - **Minimize global variables:** They can lead to naming conflicts and make code harder to maintain and debug.
  - **Use namespaces or modules:** To organize global variables and prevent conflicts.
  - **Be aware of potential pollution:** Global variables can be unintentionally modified by different parts of the code.

JavaScript

```
// Example (Use with caution!)
let appName = "MyWebApp"; // Global variable

function displayAppName() {
  console.log("App Name:", appName);
}
```

```
displayAppName();
console.log("Global App Name:", appName);
```

2. Function Scope

- **Explanation:** Variables declared with `var` inside a function have function scope. They are accessible within the entire function, including nested blocks.
- **Use Cases:**
  - Variables that are only needed within a specific function.
  - Creating private variables within a function (using closures).
- **Best Practices:**
  - **Avoid `var` if possible:** `let` and `const` provide better scope control.
  - **Use closures:** To create private variables and functions within a function.
  - **Understand hoisting:** Variables declared with `var` are hoisted to the top of the function scope, which can lead to unexpected behavior.

JavaScript

```
function myFunction() {
  var functionVar = "Function variable";

  if (true) {
    var innerVar = "Inner function variable (var)";
    console.log(innerVar); // Accessible here
  }

  console.log(functionVar); // Accessible here
  console.log(innerVar); // Accessible here due to var hoisting.
}

myFunction();
// console.log(functionVar); // Error: functionVar is not defined outside myFunction
```

3. Block Scope

- **Explanation:** Variables declared with `let` or `const` inside a block (e.g., `if`, `for`, `while`, `{}`) have block scope. They are only accessible within that block.
- **Use Cases:**
  - Variables that are only needed within a specific block of code.
  - Preventing naming conflicts between variables in different blocks.
  - Improving code clarity and maintainability.
- **Best Practices:**
  - **Prefer `let` and `const`:** Over `var` for better scope control.
  - **Use `const` when possible:** To prevent accidental reassignment of variables.
  - **Declare variables close to their use:** This improves code readability.
  - **Understand the Temporal Dead Zone (TDZ):** `let` and `const` variables are hoisted but not initialized, so accessing them before their declaration results in a `ReferenceError`.

JavaScript

```
function myBlockFunction() {
  let blockVar = "Block variable (let)";
  const blockConst = "Block constant";

  if (true) {
    let innerBlockVar = "Inner block variable (let)";
    const innerBlockConst = "Inner block constant";
    console.log(innerBlockVar);
    console.log(innerBlockConst);
  }

  console.log(blockVar);
  console.log(blockConst);
  // console.log(innerBlockVar); // Error: innerBlockVar is not defined
  // console.log(innerBlockConst); // Error: innerBlockConst is not defined
}

myBlockFunction();
```

Practical Examples and Scenarios

- **Loop Counters:** Use `let` for loop counters to limit their scope to the loop.

JavaScript

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
// console.log(i); // Error: i is not defined outside the loop
```

- **Conditional Variables:** Use `let` or `const` inside `if` statements to define variables that are only needed within that condition.

JavaScript

```
if (userIsLoggedIn) {
  const welcomeMessage = "Welcome, user!";
  console.log(welcomeMessage);
}
// console.log(welcomeMessage); // Error: welcomeMessage is not defined
```

- **Module Scope (using ES Modules):** In modern JavaScript, modules allow you to create files with their own scope, preventing global scope pollution.

JavaScript

```
// moduleA.js
export const moduleVar = "Module variable";

// moduleB.js
import { moduleVar } from "./moduleA.js";
console.log(moduleVar);
```

▼ **Hoisting:** Variable and Function hoisting

What is Hoisting?

Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope during the compilation phase. This means that you can technically use variables and functions before they are declared in your code. However, the exact behavior differs between variable and function hoisting.

1. Variable Hoisting

- `var` **Hoisting:**
  - Variables declared with `var` are hoisted to the top of their function or global scope.
  - Only the declaration is hoisted; the initialization (assignment) remains in its original place.
  - If you try to access a `var` variable before it's initialized, you'll get `undefined`.

JavaScript

```
console.log(myVar); // Output: undefined
var myVar = 10;
console.log(myVar); // Output: 10

function exampleVar() {
  console.log(innerVar); // Output: undefined
  var innerVar = 20;
  console.log(innerVar); // Output: 20
}

exampleVar();
```

- `let` **and** `const` **Hoisting:**
  - `let` and `const` variables are also hoisted, but they are not initialized.
  - Accessing them before their declaration results in a `ReferenceError` (Temporal Dead Zone - TDZ).
  - This TDZ behaviour was introduced to avoid some of the unexpected behaviours of var hoisting.

JavaScript

```
// console.log(myLet); // ReferenceError: Cannot access 'myLet' before initialization
let myLet = 30;
console.log(myLet); // Output: 30

// console.log(myConst); // ReferenceError: Cannot access 'myConst' before initialization
const myConst = 40;
console.log(myConst); // Output: 40
```

2. Function Hoisting

- **Function Declarations:**
  - Function declarations are hoisted entirely, including their implementation.
  - You can call a function declared with a function declaration before its actual declaration in the code.

JavaScript

```
myFunction(); // Output: "Function hoisted!"

function myFunction() {
```

```
  console.log("Function hoisted!");
}
```

- **Function Expressions:**
  - Function expressions are hoisted like variables.
  - If you use `var` with a function expression, the variable is hoisted but the function assignment is not.
  - If you use `let` or `const` with a function expression, then the variable is hoisted, but like normal `let` and `const` variables, it will throw a ReferenceError if called before its assignment.

JavaScript

```
// myExpression(); // TypeError: myExpression is not a function
var myExpression = function() {
  console.log("Function expression");
};

myExpression(); // Output: Function expression

// const constExpression = function(){console.log("const expression")};
// constExpression(); //ReferenceError: Cannot access 'constExpression' before initialization.
```

Best Practices and Uses

1. **Avoid `var`:**
   - Prefer `let` and `const` to avoid the unexpected behavior of `var` hoisting and the problems it can cause.
   - `let` and `const` provide better scope control and help prevent accidental variable overwrites.
2. **Declare Variables at the Top of Their Scope:**
   - Even though hoisting occurs, it's best practice to declare variables at the top of their scope.
   - This improves code readability and makes it easier to understand the flow of your code.
3. **Use Function Declarations for Hoisting:**
   - If you need to call a function before its declaration, use function declarations.
   - However, be mindful of code organization and ensure that the function's purpose is clear.
4. **Be Aware of the Temporal Dead Zone (TDZ):**
   - When using `let` and `const`, understand that they are hoisted but not initialized.
   - Avoid accessing these variables before their declaration to prevent `ReferenceError` exceptions.
5. **Function expressions are best used when functions are conditionally defined, or used as callbacks.**
6. **Use strict mode ('use strict'):** Using strict mode will help catch mistakes that can be caused by hoisting.

Example Scenarios

- **Modular Code:** In larger projects, you might use function declarations to define utility functions that are called from various parts of the code.
- **Initialization Logic:** You might use function expressions to define initialization logic that is executed only when certain conditions are met.

By understanding hoisting and following these best practices, you can write more robust and maintainable JavaScript code.

▼ **Objects and Prototypes:** Object literals, Constructors, `this` keyword, Prototypes, Inheritance (prototype chain)

Let's explore JavaScript objects and prototypes, covering object literals, constructors, the `this` keyword, prototypes, and the prototype chain, along with best practices.

1. Object Literals

- **Explanation:** Object literals are a simple way to create objects using curly braces `{}` and key-value pairs.
- **Use Cases:**
  - Creating simple data structures.
  - Defining configuration objects.
  - Representing real-world entities.

JavaScript

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 30,
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
};

console.log(person.firstName); // "John"
console.log(person.fullName()); // "John Doe"
```

In this lesson, you'll learn about accessing, adding, and updating properties of objects in JavaScript using **dot notation** and **square bracket notation**.

Accessing, Adding, and Updating an Object's Properties

Objects in JavaScript are collections of **key-value pairs**, and you can access, add, or update these properties using two main notations: **dot notation** and **square bracket notation**.

Dot Notation .

Dot notation is the most common way to **access**, **add**, or **update** properties of an object. It provides a straightforward syntax for interacting with object properties using .

Accessing Properties

To retrieve a value from an object, use **dot notation**:

```
const person = {
    firstName: 'John',
    lastName: 'Doe',
    age: 25,
};

console.log(person.firstName); // "John"
```

Adding or Updating Properties

You can also use dot notation to **add new properties** or **update existing ones**:

```
// Adding a new property
person.dog = { name: 'Fluffy', age: 2 };
```

```
// Updating an existing property
person.age = 26;

console.log(person);
// { firstName: 'John', lastName: 'Doe', age: 26, dog: { name: 'Fluffy', age: 2 } }
```

Square Bracket Notation [ ]

Square bracket notation [ ] is another way to **access properties** of an object. It offers more **flexibility**, especially when dealing with **dynamic** property names or keys that are not valid JavaScript identifiers.

Accessing Properties

You can access properties using square brackets:

```
console.log(person['firstName']); // "John"
```

Dynamic Property Access

Square bracket notation allows for **dynamic property access**, which is useful when the property name is stored in a variable:

```
const property = 'age';
console.log(person[property]); // 26
```

Handling Unusual Key Names

Square bracket notation is also necessary when dealing with keys that contain **spaces**, **dashes**, or other characters **not allowed** in JavaScript identifiers:

```
// Adding properties with unusual key names
person['this-is-a-key-with-dashes'] = 'value1';
person['this is another key'] = 'value2';

console.log(person['this-is-a-key-with-dashes']); // "value1"
console.log(person['this is another key']); // "value2"

// Using dot notation with these keys would result in an error
// person.this-is-a-key-with-dashes - error
// person.this is another key - error
```

Understanding how to work with **object properties** is crucial for effectively **managing data** in JavaScript applications. **Dot** notation . and **square bracket** [ ] notation each have their use cases, and knowing when to use each will enhance your coding skills.

Object Methods

What is a Method?

A **method** is a **function** associated with an **object**. Simply put, a method is a property of an object that is a function. Methods allow objects to **perform actions** and are defined similarly to regular functions, but they are assigned as **properties** of an object.

Defining Methods

Methods can be defined in two main ways:

Assigning a Function to an Object Property

- **Assigning a Function to an Object Property**

```
var myObj = {
  myMethod: function(params) {
    // ...do something
  },

  // OR using shorthand syntax
  myOtherMethod(params) {
    // ...do something else
  }
};
```

In this example, myMethod and myOtherMethod are methods of myObj.

Using a Function as a Method

- **Using a Function as a Method**

You can assign a function to an object property, making it a method:

```
objectName.methodname = functionName;
```

Where objectName is an existing object, methodname is the name you are assigning to the method, and functionName is the name of the function.

Calling Methods

You can call a method in the **context** of the object as follows:

```
object.methodname(params);
```

Defining Methods for an Object Type

You can **define** methods for an object type by including a **method definition** in the **object constructor** function.

For example, let's define a function that formats and displays the properties of Car objects:

```
function displayCar() {
  var result = `A Beautiful ${this.year} ${this.make} ${this.model}`;
  pretty_print(result);
}
```

Here, pretty_print is a function to display a formatted string. Notice the use of this to refer to the object to which the method belongs.

Adding Methods to an Object Constructor

You can make this function a method of Car by adding the statement:

```
this.displayCar = displayCar;
```

So, the full definition of Car would now look like this:

```
function Car(make, model, year, owner) {
  this.make = make;
```

```
  this.model = model;
  this.year = year;
  this.owner = owner;
  this.displayCar = displayCar;
}
```

Calling Methods on Objects

You can **call** the displayCar method for each of the Car objects as follows:

```
var car1 = new Car('Toyota', 'Corolla', 2020, 'Alice');
var car2 = new Car('Honda', 'Civic', 2019, 'Bob');

car1.displayCar();
car2.displayCar();
```

Methods are a powerful feature in JavaScript, enabling objects to have **behaviors** and **perform actions**. Understanding how to define and use methods is essential for working with objects and building complex applications.

Built-in Object Methods

Objects in JavaScript are collections of **key-value pairs**, capable of holding various data types, including **strings**, **numbers**, and **booleans**. All objects in JavaScript **inherit from the parent** Object constructor, which offers several built-in methods to simplify object manipulation. Unlike array methods like sort() and reverse(), which are used on array instances, object methods are **static** and used directly on the Object constructor, taking the object instance as a **parameter**.

Key Built-in Object Methods

Object.create()

The Object.create() method creates a new object **linked** to the prototype of an existing object. This is useful for **extending** objects **without duplicating** code.

```
const role = {
    title: 'developer',
    type: 'full-time',
    isOpen: true,
    showDetails() {
        const status = this.isOpen ? 'is open for applications' : "is not open for applications";
        console.log(`The ${this.title} role is ${this.type} and ${status}.`);
    }
};

const designer = Object.create(role);
designer.title = "designer";
designer.showDetails(); // The designer role is full-time and is open for applications.
```

Object.keys()

Object.keys() returns an array of an object's keys, allowing you to iterate over them or check their count.

```
const team = {
    leader: 'Alice',
    developer: 'Bob',
    designer: 'Charlie',
```

```
    tester: 'Dana'
};

const keys = Object.keys(team);
console.log(keys); // ["leader", "developer", "designer", "tester"]

keys.forEach(key => {
    console.log(`${key}: ${team[key]}`);
});
```

Object.values()

Object.values() returns an array of an object's values, useful for iterating over or analyzing data.

```
const device = {
    brand: 'Samsung',
    model: 'Galaxy S21',
    year: 2021
};

const values = Object.values(device);
console.log(values); // ["Samsung", "Galaxy S21", 2021]
```

Object.entries()

Object.entries() returns a nested array of an object's key-value pairs, allowing for easy iteration and manipulation.

```
const software = {
    name: 'Photoshop',
    version: '2021',
    license: 'Commercial'
};

const entries = Object.entries(software);
entries.forEach(([key, value]) => {
    console.log(`${key}: ${value}`);
});
```

Object.assign()

Object.assign() copies values from one object to another, useful for merging objects.

```
const personalDetails = { firstName: 'Jane', lastName: 'Doe' };
const jobDetails = { position: 'Manager', company: 'Tech Inc' };

const profile = Object.assign({}, personalDetails, jobDetails);
console.log(profile); // {firstName: "Jane", lastName: "Doe", position: "Manager", company: "Tech Inc"}
```

Object.freeze()

Object.freeze() prevents modifications to an object's properties and values, ensuring immutability.

```
const settings = { theme: 'light', notifications: true };
Object.freeze(settings);
```

```
settings.theme = 'dark'; // No effect
console.log(settings); // {theme: "light", notifications: true}
```

Object.seal()

Object.seal() prevents new properties from being added to an object but allows modification of existing properties.

```
const account = { username: 'user123', password: 'pass123' };
Object.seal(account);

account.password = 'newpass'; // Allowed
account.email = 'user@example.com'; // Not allowed
console.log(account); // {username: "user123", password: "newpass"}
```

Object.getPrototypeOf()

Object.getPrototypeOf() retrieves the prototype of an object, useful for understanding inheritance and prototype chains.

```
const gadgets = ['laptop', 'tablet', 'smartphone'];
console.log(Object.getPrototypeOf(gadgets) === Array.prototype); // true
```

Conclusion

JavaScript's **built-in object methods** provide essential tools for **managing** and **manipulating** objects, allowing you to **create**, **modify**, and **protect** data effectively. Understanding these methods enhances your ability to work with objects and leverage their full potential in your applications.

2. Constructors

- **Explanation:** Constructor functions are used to create multiple objects with similar properties and methods.
- **Use Cases:**
  - Creating multiple instances of an object.
  - Initializing object properties.
  - Defining object behavior.

JavaScript

```
function Person(firstName, lastName, age) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;
  this.fullName = function() {
    return this.firstName + " " + this.lastName;
  };
}

const person1 = new Person("Jane", "Smith", 25);
const person2 = new Person("David", "Lee", 35);

console.log(person1.fullName()); // "Jane Smith"
console.log(person2.age); // 35
```

3. The `this` Keyword

- **Explanation:** The `this` keyword refers to the object that is currently executing the function. Its value depends on how the function is called.
- **Use Cases:**
  - Accessing object properties within methods.
  - Referring to the current object.
  - Handling event listeners.

JavaScript

```
const car = {
  make: "Toyota",
  model: "Camry",
  start: function() {
    console.log("Starting " + this.make + " " + this.model);
  }
};

car.start(); // "Starting Toyota Camry"

function globalFunction() {
    console.log(this); // in a browser, this will be the window object. In node.js, it will be the global object.
}
globalFunction();
```

- **Arrow Functions and `this`:** Arrow functions do not have their own `this` context. They inherit `this` from the surrounding scope (lexical `this`).

4. Prototypes

- **Explanation:** Prototypes are objects from which other objects inherit properties and methods. Every JavaScript object has a prototype.
- **Use Cases:**
  - Sharing methods between objects.
  - Implementing inheritance.
  - Extending built-in objects.

JavaScript

```
function Animal(name) {
  this.name = name;
}

Animal.prototype.sayHello = function() {
  console.log("Hello, I am " + this.name);
};

const dog = new Animal("Buddy");
dog.sayHello(); // "Hello, I am Buddy"
```

5. Inheritance (Prototype Chain)

- **Explanation:** The prototype chain is a sequence of prototype objects, where each object inherits properties and methods from its prototype.

- **Use Cases:**
  - Creating hierarchical relationships between objects.
  - Reusing code through inheritance.
  - Implementing polymorphism.

JavaScript

```
function Dog(name, breed) {
  Animal.call(this, name); // Call the Animal constructor
  this.breed = breed;
}

Dog.prototype = Object.create(Animal.prototype); // Inherit from Animal

Dog.prototype.bark = function() {
  console.log("Woof!");
};

const myDog = new Dog("Max", "Golden Retriever");
myDog.sayHello(); // "Hello, I am Max" (inherited from Animal)
myDog.bark(); // "Woof!"
```

Best Practices

1. **Use `class` Syntax (ES6):** The `class` syntax provides a more structured and readable way to define objects and inheritance.

JavaScript

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  sayHello() {
    console.log(`Hello, I am ${this.name}`);
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }

  bark() {
    console.log("Woof!");
  }
}

const myDog = new Dog("Max", "Golden Retriever");
myDog.sayHello();
myDog.bark();
```

1. **Avoid Modifying Built-in Prototypes:** Modifying built-in prototypes can lead to unexpected behavior and conflicts.
2. **Use `Object.create()` for Prototype Inheritance:** Use `Object.create()` to create a new object with the specified prototype.
3. **Understand `this` Context:** Be aware of how `this` behaves in different contexts, especially with arrow functions and event handlers.
4. **Favor Composition Over Inheritance:** In some cases, composition (combining objects) can be a more flexible and maintainable approach than inheritance.
5. **Use `instanceof` to Check Object Types:** The `instanceof` operator checks if an object is an instance of a specific constructor.

JavaScript

```
console.log(myDog instanceof Dog); // true
console.log(myDog instanceof Animal); // true
```

By understanding objects and prototypes, you can write more organized and efficient JavaScript code.

▼ **Type Coercion:** Understanding how JavaScript converts types automatically

How Type Coercion Works

JavaScript tries to convert values to the type it expects in certain contexts. Here are some common scenarios:

1. **String Concatenation (+):**
   - If one operand is a string, JavaScript converts the other operand to a string and concatenates them.
2. **Arithmetic Operations (-, *, /):**
   - JavaScript attempts to convert operands to numbers. If it fails, it results in `NaN`.
3. **Comparison Operators (==, !=, <, >, <=, >=):**
   - JavaScript tries to convert operands to a common type before comparison. The `==` and `!=` operators are particularly prone to implicit type coercion.
4. **Logical Operators (&&, ||, !):**
   - JavaScript converts values to booleans (truthy or falsy) for logical operations.

Sample Code

JavaScript

```
// String Coercion
console.log("5" + 3); // "53" (3 is converted to "3")
console.log(1 + "2"); // "12" (1 is converted to "1")
console.log(1 + 2 + "3"); // "33" (1+2 = 3, then 3 is converted to "3")

// Numeric Coercion
console.log("5" - 3); // 2 ("5" is converted to 5)
console.log("10" * "2"); // 20 ("10" and "2" are converted to numbers)
console.log("hello" * 2); // NaN ("hello" cannot be converted to a number)
console.log(10 / "2"); // 5

// Boolean Coercion (Truthy and Falsy)
console.log(1 == true); // true (1 is converted to true)
console.log(0 == false); // true (0 is converted to false)
console.log("" == false); // true ("" is converted to false)
console.log(null == undefined); // true
console.log(null === undefined); // false
console.log(0 == ""); // true
```

```
console.log(0 === ""); // false
console.log(1 && "hello"); // "hello" (1 is truthy)
console.log(0 || "world"); // "world" (0 is falsy)
console.log(!""); // true ("" is falsy, !false is true)
console.log(!0); // true (0 is falsy, !false is true)

// More examples
console.log([] == false); // true. Empty array is coerced to zero, then false.
console.log([] == ![]); // false. ![] is false, and empty array is coerced to zero, then false.

// Type coercion with if statements.
if ("0") {
  console.log("Truthy"); // "Truthy", because "0" is truthy.
}

if (0) {
  console.log("Falsy"); // This will not execute because 0 is falsy.
}
```

Truthy and Falsy Values

In JavaScript, some values are considered "truthy" (they evaluate to `true` in a boolean context), and others are "falsy" (they evaluate to `false`).

- **Falsy Values:**
  - `false`
  - `0` (zero)
  - `""` (empty string)
  - `null`
  - `undefined`
  - `NaN`
- **Truthy Values:**
  - Any value that is not falsy (e.g., `"hello"`, `1`, `[]`, `{}`)

Best Practices

1. **Use Strict Equality (===, !==):**
   - Prefer `===` and `!==` over `==` and `!=` to avoid implicit type coercion. This makes your code more predictable and reduces the risk of unexpected behavior.
2. **Explicit Type Conversion:**
   - When you need to convert a value to a specific type, do it explicitly using functions like:
     - `Number()`
     - `String()`
     - `Boolean()`
     - `parseInt()` or `parseFloat()`
3. **Be Mindful of Truthy/Falsy Values:**
   - Understand which values are truthy and falsy, and use them intentionally in conditional statements.
4. **Avoid Unnecessary Coercion:**

- Write code that minimizes the need for implicit type coercion. This makes your code easier to read and understand.

5. **Use `typeof` and `instanceof`:**
   - When you need to check the type of a variable, use the `typeof` operator or the `instanceof` operator.

Example of Explicit Type Conversion

JavaScript

```
let str = "10";
let num = Number(str); // Explicit conversion to number

if (num === 10) {
  console.log("Number is 10");
}

let value = 1;
let bool = Boolean(value); // Explicit conversion to boolean

if (bool) {
  console.log("Value is truthy");
}
```

By understanding type coercion and following these best practices, you can write more robust and predictable JavaScript code.

II. Intermediate (Core Concepts)

▼ **Arrays:** Array methods (`push`, `pop`, `shift`, `unshift`, `map`, `filter`, `reduce`, `forEach`, etc.)

What are Arrays?

Arrays are ordered collections of values. They can hold elements of any data type (numbers, strings, objects, etc.). Arrays are zero-indexed, meaning the first element is at index 0.

Common Array Methods

1. `push()`:JavaScript
   - **Explanation:** Adds one or more elements to the *end* of an array.
   - **Use:** Appending items to a list.
   - **Example:**

```
let numbers = [1, 2, 3];
numbers.push(4, 5);
console.log(numbers); // [1, 2, 3, 4, 5]
```

2. `pop()`:JavaScript
   - **Explanation:** Removes the *last* element from an array and returns it.
   - **Use:** Removing the last item from a list.
   - **Example:**

```
let numbers = [1, 2, 3];
let last = numbers.pop();
console.log(numbers); // [1, 2]
console.log(last); // 3
```

3. `shift()`:JavaScript
   - **Explanation:** Removes the *first* element from an array and returns it.
   - **Use:** Removing the first item from a queue.

- **Example:**

```
let numbers = [1, 2, 3];
let first = numbers.shift();
console.log(numbers); // [2, 3]
console.log(first); // 1
```

4. `unshift()` :JavaScript
   - **Explanation:** Adds one or more elements to the *beginning* of an array.
   - **Use:** Adding items to the front of a queue.
   - **Example:**

```
let numbers = [2, 3];
numbers.unshift(0, 1);
console.log(numbers); // [0, 1, 2, 3]
```

5. `map()` :JavaScript
   - **Explanation:** Creates a *new* array by applying a function to each element of the original array.
   - **Use:** Transforming array elements.
   - **Example:**

```
let numbers = [1, 2, 3];
let doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6]
```

6. `filter()` :JavaScript
   - **Explanation:** Creates a *new* array with elements that pass a test (a function that returns `true` or `false`).
   - **Use:** Selecting elements that meet certain criteria.
   - **Example:**

```
let numbers = [1, 2, 3, 4, 5];
let even = numbers.filter(num => num % 2 === 0);
console.log(even); // [2, 4]
```

7. `reduce()` :JavaScript
   - **Explanation:** Reduces an array to a single value by applying a function to each element and an accumulator.
   - **Use:** Calculating sums, averages, or other aggregate values.
   - **Example:**

```
let numbers = [1, 2, 3, 4, 5];
let sum = numbers.reduce((acc, num) => acc + num, 0); // 0 is the initial accumulator value.
console.log(sum); // 15
```

8. `forEach()` :JavaScript
   - **Explanation:** Executes a function for each element in an array. Does not create a new array.
   - **Use:** Iterating over array elements to perform actions.
   - **Example:**

```
let numbers = [1, 2, 3];
numbers.forEach(num => console.log(num)); // 1, 2, 3
```

9. `slice()` :JavaScript
   - **Explanation:** Creates a new array containing a portion of the original array.
   - **Use:** Extracting a subarray.
   - **Example:**

```
let numbers = [1, 2, 3, 4, 5];
let subArray = numbers.slice(1, 4); // start index, end index(not included)
console.log(subArray); // [2, 3, 4]
```

10. `splice()` :JavaScript
    - **Explanation:** Changes the contents of an array by removing or replacing existing elements and/or adding new elements in place. 1. github.com

      1

      github.com
    - **Use:** Inserting, deleting, or replacing elements.
    - **Example:**

```
let numbers = [1, 2, 3, 4, 5];
numbers.splice(2, 2, 6, 7); // start index, delete count, new elements
console.log(numbers); // [1, 2, 6, 7, 5]
```

11. `concat()` :JavaScript
    - **Explanation:** Merges two or more arrays into a new array.
    - **Use:** Combining arrays.
    - **Example:**

```
let array1 = [1, 2];
let array2 = [3, 4];
let combined = array1.concat(array2);
console.log(combined); // [1, 2, 3, 4]
```

    1. `indexOf()` **and** `lastIndexOf()` :JavaScript
       - **Explanation:** Returns the first or last index at which a given element can be found in the array, or -1 if it is not present.
       - **Use:** Finding an element's position.
       - **Example:**

```
let numbers = [1, 2, 3, 2, 4];
console.log(numbers.indexOf(2)); // 1
console.log(numbers.lastIndexOf(2)); // 3
```

12. `find()` **and** `findIndex()` :JavaScript

- **Explanation:** `find()` returns the first element that satisfies the provided testing function. `findIndex()` returns the index of the first element that satisfies the provided testing function.
- **Use:** Finding a specific object, or the index of a specific object in an array.
- **Example:**

```
let people = [{name: "Alice", age: 25}, {name: "Bob", age: 30}, {name: "Charlie", age: 25}];
let foundPerson = people.find(person => person.age === 30);
console.log(foundPerson); // {name: "Bob", age: 30}
console.log(people.findIndex(person => person.age === 25)); // 0
```

Best Practices

- **Immutability:** When possible, use methods like `map()`, `filter()`, and `reduce()` that create new arrays instead of modifying the original array. This helps avoid side effects.
- **Clarity:** Choose the appropriate method for the task. `forEach()` for side effects, `map()` for transformations, `filter()` for selections, and `reduce()` for aggregations.
- **Chaining:** Chain array methods to perform multiple operations in a concise way.
- **Readability:** Write clear and concise code that is easy to understand.
- **Avoid `for...in` for Arrays:** Use `for...of` or `forEach()` for iterating over array elements. `for...in` is intended for iterating over object properties.
- **Use `const` with Array Methods:** If you are not reassigning the array, use `const` when declaring it.
- **Use `find()` and `findIndex()` for object arrays:** They are very useful when dealing with arrays of complex objects.

▼ **DOM Manipulation (Browser Environment):** Selecting elements (e.g., `getElementById`, `querySelector`), Modifying content, Attributes, Styling, Event listeners

What is the DOM?

The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects.

1. Selecting Elements

- `getElementById()`: Selects an element by its `id` attribute.JavaScript

```
const myElement = document.getElementById("myId");
```

- `getElementsByClassName()`: Selects elements by their `class` attribute. Returns an HTMLCollection (an array-like object).JavaScript

```
const elements = document.getElementsByClassName("myClass");
```

- `getElementsByTagName()`: Selects elements by their tag name. Returns an HTMLCollection.JavaScript

```
const paragraphs = document.getElementsByTagName("p");
```

- `querySelector()`: Selects the *first* element that matches a CSS selector.JavaScript

```
const myElement = document.querySelector("#myId"); // Selects by ID
const firstParagraph = document.querySelector("p"); // Selects the first paragraph
const elementWithClass = document.querySelector(".myClass"); //Selects first element with class.
```

- `querySelectorAll()`: Selects *all* elements that match a CSS selector. Returns a NodeList (an array-like object).JavaScript

```
const allParagraphs = document.querySelectorAll("p");
const allElementsWithClass = document.querySelectorAll(".myClass");
```

2. Modifying Content

- `textContent`: Gets or sets the text content of an element.JavaScript

```
myElement.textContent = "New text content";
console.log(myElement.textContent);
```

- `innerHTML`: Gets or sets the HTML content of an element.JavaScript

```
myElement.innerHTML = "<strong>Bold text</strong>";
```

- `createElement()`: Creates a new element node.

const newParagraph = document.createElement("p");
newParagraph.textContent = "This is a new paragraph.";
document.body.appendChild(newParagraph); // Add to the document
```

- `appendChild()`: Adds a node as the last child of an element.
- `removeChild()`: Removes a child node from an element.JavaScript

```
document.body.removeChild(newParagraph);
```

- `replaceChild()`: Replaces a child node with a new node.JavaScript

```
const newHeading = document.createElement("h1");
newHeading.textContent = "New Heading";
document.body.replaceChild(newHeading, myElement);
```

3. Modifying Attributes

- `getAttribute()`: Gets the value of an attribute.JavaScript

```
const link = document.querySelector("a");
const href = link.getAttribute("href");
console.log(href);
```

- `setAttribute()`: Sets the value of an attribute.JavaScript

```
link.setAttribute("href", "https://www.example.com");
```

- `removeAttribute()`: Removes an attribute.JavaScript

```
link.removeAttribute("target");
```

- `classList`: Provides methods for working with an element's class attribute.JavaScript

```
myElement.classList.add("active");
myElement.classList.remove("active");
myElement.classList.toggle("active");
myElement.classList.contains("active");
```
```

4. Modifying Styling

- `style` **property:** Sets inline styles.JavaScript

```
myElement.style.color = "red";
myElement.style.backgroundColor = "lightblue";
```

- `classList` **:** Adds or removes CSS classes.JavaScript

```
myElement.classList.add("highlight"); // Assuming .highlight is defined in CSS
```

5. Event Listeners

- `addEventListener()` **:** Attaches an event handler to an element.JavaScript

```
const button = document.querySelector("button");
button.addEventListener("click", function() {
  console.log("Button clicked!");
});

button.addEventListener("mouseover", function() {
    button.style.backgroundColor = "yellow";
});

button.addEventListener("mouseout", function() {
    button.style.backgroundColor = "";
});
```

- **Event Types:**
  - `click`
  - `mouseover`
  - `mouseout`
  - `keydown`
  - `keyup`
  - `submit`
  - `load`
  - `change`
  - And many more...
- **Removing Event Listeners:**JavaScript

```
function handleClick() {
  console.log("Button clicked!");
  button.removeEventListener("click", handleClick); //remove the event listener after the first click.
}
button.addEventListener("click", handleClick);
```

Best Practices

- **Minimize DOM Manipulation:** DOM operations can be expensive. Try to minimize the number of times you modify the DOM.
- **Use Event Delegation:** Attach event listeners to parent elements rather than individual child elements.

- **Use `classList` :** For adding and removing CSS classes, `classList` is more efficient and readable than directly manipulating the `className` property.
- **Separate Concerns:** Keep your JavaScript code separate from your HTML and CSS.
- **Use `querySelector()` and `querySelectorAll()` :** They are generally more flexible and powerful than the older `getElementById()` and `getElementsByClassName()` methods.
- **Use `textContent` when only dealing with text:** `textContent` is faster than `innerHTML` when you are just setting text.
- **Be aware of reflows and repaints:** Changes to the DOM can cause the browser to recalculate layout (reflow) and repaint the screen. Batch DOM changes to reduce these operations.
- **Use `requestAnimationFrame()` for animations:** When animating, use `requestAnimationFrame()` to synchronize animations with the browser's refresh rate.

HTML Attributes and DOM Properties

HTML elements can have various **attributes** assigned to them, such as id, class, type, and more. These attributes define **characteristics** and **behaviors** of elements in an HTML document.

Attributes vs. Properties

- **Attributes**: These are defined in the HTML markup and specify initial values for elements.
- **Properties**: These are part of the DOM objects and represent the current state of elements.

When a browser processes an **HTML document**, it creates corresponding **DOM properties** for standard attributes. Some attributes apply to **all elements**, while others are **specific** to certain elements.

For example, id and class apply to **all elements**, while type is **specific** to inputs and buttons.

Common Properties

- classList: A list of classes assigned to an element. It's an array-like object.
- className: A string of classes assigned to an element, separated by spaces if there are multiple classes.
- id: A string representing the ID assigned to an element.
- innerHTML: The inner content of the element, including nested elements, in string form.

Common Methods

- addEventListener: Listens for a specified event and calls a function when that event occurs. Events can include click, mousedown, mouseup, focus, blur, etc.
- getBoundingClientRect: Returns the height, width, left, and top values of an element relative to the browser.
- hasAttribute: Checks if an element has a specific attribute.
- removeAttribute: Removes a specified attribute from an element.
- removeEventListener: Removes an event listener from an element.
- scroll: Scrolls to an element's position.

*Example **

Here's an example demonstrating the use of some **properties** and **methods**:

```
<div id="div1" class="class1 class2"><span>hello</span></div>
<input id="input" type="text">
<button>Button</button>

<script>
  var element = document.getElementById("div1");
```

```
  console.log(element.classList); // ["class1", "class2"]
  console.log(element.className); // "class1 class2"
  console.log(element.id); // "div1"
  console.log(element.innerHTML); // "<span>hello</span>"

  // Example of using a method
  element.addEventListener('click', function() {
    alert('Div clicked!');
  });
</script>
```

Conclusion

For more detailed information on DOM properties and methods, you can refer to the MDN Web Docs. The MDN documentation provides comprehensive guides and examples to help you understand and use these features effectively.

Understanding the **difference between attributes and properties**, as well as how to use common DOM methods, is crucial for effectively **manipulating** and **interacting** with HTML elements in JavaScript.

▼ **Events:** Event types (click, submit, keypress, etc.), Event handling, Event propagation (bubbling, capturing)

What are Events?

Events are signals or notifications that something has happened in the browser. They represent user interactions (like clicks or key presses), browser actions (like page loading), or other occurrences. Events provide a way for JavaScript to respond to these actions and make web pages interactive.

Event Types (Examples)

- **Mouse Events:**
  - `click`: User clicks a mouse button.
  - `dblclick`: User double-clicks a mouse button.
  - `mousedown` / `mouseup`: Mouse button is pressed/released.
  - `mouseover` / `mouseout`: Mouse pointer enters/leaves an element.
  - `mousemove`: Mouse pointer moves within an element.
- **Keyboard Events:**
  - `keydown`: Key is pressed.
  - `keyup`: Key is released.
  - `keypress`: Key is pressed and released (less used now).
- **Form Events:**
  - `submit`: Form is submitted.
  - `change`: Value of a form element changes.
  - `focus` / `blur`: Element gains/loses focus.
  - `input`: Value of a text input changes.
- **Document/Window Events:**
  - `load`: Page and all resources are loaded.
  - `DOMContentLoaded`: Initial HTML is loaded and parsed.
  - `resize`: Window is resized.

- `scroll`: Document is scrolled.

- **Touch Events (Mobile):**
  - `touchstart`, `touchmove`, `touchend`, `touchcancel`: Touch interactions.

Event Handling

Event handling involves attaching functions (event listeners) to elements to respond to specific events.

- `addEventListener()`: The standard and preferred method.JavaScript

```
const button = document.querySelector("#myButton");

button.addEventListener("click", function(event) {
  console.log("Button clicked!");
  console.log(event); // The event object
});
```

  - `event` parameter: An event object with details about the event.

- **Inline Event Handlers (Avoid):**HTML

```
<button onclick="console.log('Click!')">Click</button>
```

  - Avoid inline handlers due to separation of concerns.

- **Removing Event Listeners:**JavaScript

```
function handleClick() {
  console.log("Clicked!");
  button.removeEventListener("click", handleClick);
}

button.addEventListener("click", handleClick);
```

Event Propagation

Event propagation describes how events travel through the DOM tree.

- **Event Bubbling (Default):**
  - Event starts at the target element and travels *up* the DOM tree to ancestor elements.
- **Event Capturing:**
  - Event starts at the root element and travels *down* the DOM tree to the target element.

HTML

```
<div id="outer">
  <div id="inner">
    <button id="myButton">Click</button>
  </div>
</div>
```

JavaScript

```
document.getElementById("outer").addEventListener("click", function() {
  console.log("Outer (bubbling)");
});
```

```
document.getElementById("inner").addEventListener("click", function() {
  console.log("Inner (bubbling)");
});

document.getElementById("myButton").addEventListener("click", function() {
  console.log("Button (bubbling)");
});

//Capturing
document.getElementById("outer").addEventListener("click", function() {
    console.log("Outer (capturing)");
}, true);

document.getElementById("inner").addEventListener("click", function() {
    console.log("Inner (capturing)");
}, true);

document.getElementById("myButton").addEventListener("click", function() {
    console.log("Button (capturing)");
}, true);
```

- `event.stopPropagation()`: Stops event propagation.JavaScript

```
document.getElementById("inner").addEventListener("click", function(event) {
  console.log("Inner clicked, but stop!");
  event.stopPropagation();
});
```

- `event.preventDefault()`: Prevents the default action of an event.JavaScript

```
const link = document.querySelector("a");
link.addEventListener("click", function(event) {
  event.preventDefault(); // Prevents link navigation
  console.log("Link click prevented.");
});
```

Uses

- **User Interaction:** Responding to clicks, form submissions, and keyboard input.
- **Dynamic Content:** Updating page content based on user actions.
- **Form Validation:** Checking form data before submission.
- **Animations:** Triggering animations based on events.
- **Game Development:** Handling user input in browser-based games.
- **Accessibility:** Providing keyboard navigation and other accessibility features.

Best Practices

- **Use** `addEventListener()`: For attaching event listeners.
- **Separate Concerns:** Keep JavaScript separate from HTML.
- **Event Delegation:** Attach listeners to parent elements for efficiency.JavaScript

```
document.querySelector("#myList").addEventListener("click", function(event) {
  if (event.target.tagName === "LI") {
    console.log("List item clicked:", event.target.textContent);
  }
});
```

- **Understand Propagation:** Know how events bubble and capture.
- **Use `event.stopPropagation()` and `event.preventDefault()`:** When needed.
- **Clean Up Listeners:** Remove listeners when no longer needed.
- **Use descriptive function names:** For event handlers.
- **Throttle/Debounce:** To limit the number of times an event handler is called (especially for events like `scroll` or `resize`).
- **Test event handlers:** Make sure your event handlers work as expected.
- **Use passive event listeners:** To improve scroll performance, use passive event listeners when you don't call preventDefault.

▼ **Intervals and Timers**

Welcome to a new module in this course! I am super excited for this one, because now we're going to understand some important concepts in JavaScript. These concepts will later help us provide functionalities to our app like **fetching data** and **sending data** to servers. Until now, in this course, we have been dealing with something called **synchronous JavaScript**, and it's now time to understand asynchronous JavaScript. You might not understand these terms yet, but by the end of this lecture, you will know what they are and why these concepts are crucial to building real-world apps. So, let's get started!

In this lesson, you'll learn about asynchronous JavaScript, a crucial concept for building responsive and efficient applications. **Asynchronous JavaScript** allows you to perform tasks like fetching data from a server or handling user interactions without blocking the execution of other code.

Introduction to Asynchronous JavaScript

JavaScript provides several **built-in functions** that enable you to execute code at specified **intervals** or after a **delay**, even while other code is running. This is particularly useful in scenarios like updating a game screen or tracking time on a website.

setInterval()

The setInterval function allows you to **execute** a block of code **repeatedly** at specified **time intervals**. For example, the following code logs "Hello World" every thousand milliseconds:

```
setInterval(() => {
  console.log('Hello World');
}, 1000);
```

To prevent an interval from running **indefinitely**, you can store it in a variable and clear it using the clearInterval() function:

```
const myInterval = setInterval(() => console.log("Hello World"), 1000);

// To stop the interval
clearInterval(myInterval);
```

The clearInterval function is useful when you want the interval to stop after a certain condition is met.

setTimeout()

The setTimeout function allows you to **execute** a block of code after a specified **delay**. Unlike setInterval, it only runs **once** unless set up to repeat. Here's how you use it:

```
const myTimeout = setTimeout(() => console.log('Hello, World'), 1000); // logs "Hello World" after a thousand ms

// To cancel the timeout
clearTimeout(myTimeout);
```

The clearTimeout function can be used to cancel the timeout before it executes.

Understanding Asynchronous Execution

This lesson introduces you to JavaScript code that doesn't execute **linearly** from top to bottom. Instead, it is **asynchronous**, meaning that some code can be executed after other code, even if it appears earlier in the file. This is a fundamental concept that allows JavaScript to handle tasks like **network requests** and **user interactions** efficiently.

As we continue through the course, you'll delve deeper into asynchronous JavaScript, exploring concepts like **promises** and **async/await**, which provide more **control** and **readability** when working with asynchronous operations. Understanding these concepts is essential for building real-world applications that are both **responsive** and **efficient**.

▼ **Asynchronous JavaScript:** Callbacks, Promises (`.then`, `.catch`, `.finally`), Async/Await

What is Asynchronous JavaScript?

JavaScript is single-threaded, meaning it executes code line by line. Asynchronous operations allow you to perform tasks that take time (like fetching data from a server or reading a file) without blocking the main thread, ensuring the user interface remains responsive.

1. Callbacks

- **Explanation:** Callbacks are functions that are passed as arguments to other functions and are executed when an asynchronous operation completes.
- **Use:** Handling the results of asynchronous operations.
- **Drawbacks:** Can lead to "callback hell" (nested callbacks), making code hard to read and maintain.

JavaScript

```
function fetchData(callback) {
  setTimeout(function() {
    const data = "Data fetched!";
    callback(data); // Call the callback with the data
  }, 1000);
}

fetchData(function(result) {
  console.log(result); // "Data fetched!"
});
```

2. Promises

- **Explanation:** Promises are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value.
- **States:**
  - `pending`: Initial state, neither fulfilled nor rejected.
  - `fulfilled`: Operation completed successfully.
  - `rejected`: Operation failed.

- **Methods:**
  - `.then()`: Handles the fulfilled state.
  - `.catch()`: Handles the rejected state.
  - `.finally()`: Executes code regardless of the promise's outcome.

JavaScript

```
function fetchDataPromise() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      const success = true; // Simulate success or failure
      if (success) {
        resolve("Data fetched!");
      } else {
        reject("Failed to fetch data!");
      }
    }, 1000);
  });
}

fetchDataPromise()
  .then(function(result) {
    console.log(result); // "Data fetched!"
  })
  .catch(function(error) {
    console.error(error); // "Failed to fetch data!"
  })
  .finally(function() {
    console.log("Operation complete.");
  });
```

3. Async/Await

- **Explanation:** `async` and `await` are syntactic sugar built on top of promises, making asynchronous code look and behave more like synchronous code.
- `async`: Declares a function as asynchronous, allowing the use of `await`.
- `await`: Pauses the execution of an `async` function until a promise is resolved.

JavaScript

```
async function fetchDataAsync() {
  try {
    const result = await fetchDataPromise(); // Wait for the promise to resolve
    console.log(result); // "Data fetched!"
  } catch (error) {
    console.error(error); // "Failed to fetch data!"
  } finally {
     console.log("Async operation complete.")
  }
}

fetchDataAsync();
```

Practical Uses

- **Fetching Data from APIs:**
  - Using `fetch()` or `axios` to retrieve data from a server.
- **File I/O (Node.js):**
  - Reading and writing files asynchronously.
- **Timers:**
  - Using `setTimeout()` and `setInterval()`.
- **Event Handling:**
  - Handling user interactions and other events.
- **Database Operations (Node.js):**
  - Querying and updating databases.

Best Practices

- **Use Promises or Async/Await:** Avoid callback hell by using promises or async/await.
- **Handle Errors:** Always include `.catch()` or `try...catch` blocks to handle errors gracefully.
- **Use `.finally()`:** To execute cleanup code regardless of the promise's outcome.
- **Chain Promises:** Use `.then()` to chain multiple asynchronous operations.
- **Use `Promise.all()`:** To execute multiple promises concurrently.
- **Use `Promise.race()`:** To return the result of the first promise that resolves.
- **Avoid Mixing Callbacks and Promises/Async/Await:** Stick to one style for consistency.
- **Use descriptive variable names:** To make your asynchronous code easier to read.
- **Test Asynchronous Code:** Use testing frameworks that support asynchronous testing (e.g., Jest).
- **Understand the Event Loop:** A crucial aspect of how asynchronous JavaScript works.
- **Use async/await with caution inside loops:** If you are awaiting inside a loop, each iteration will wait for the previous one to complete. If you need to run them concurrently, use `Promise.all()`.

Sources and related content

steemit.com

steemit.com

medium.com

medium.com

In this lesson, you'll learn about the differences between **synchronous** and **asynchronous** JavaScript. Understanding these concepts is crucial for writing **efficient** and **responsive** code, especially when dealing with tasks like **data fetching** and **user interactions**.

Synchronous JavaScript

Synchronous JavaScript is code that is executed **line by line**, with each task completing **instantly**. There is **no delay** in the execution of tasks for these lines of code.

Here's an example of synchronous JavaScript:

```
const functionOne = () => {
  console.log('Function One');

  functionTwo();
```

```
    console.log('Function One, Part 2');
}

const functionTwo = () => {
    console.log('Function Two');
}

functionOne();

// Output:
// Function One
// Function Two
// Function One, Part 2
```

- The code executes in a **straightforward** manner.
- First, 'Function One' is logged.
- Then, functionTwo is invoked, logging 'Function Two'.
- Finally, back in functionOne, 'Function One,Part 2' is logged.
- The execution is **linear** and **predictable**.

Asynchronous JavaScript

Now, let's explore **asynchronous** JavaScript by modifying the functions:

```
const functionOne = () => {
    console.log('Function One'); // 1

    functionTwo();

    console.log('Function One, Part 2'); // 2
}

const functionTwo = () => {
    setTimeout(() => console.log('Function Two'), 2000); // 3
}

functionOne();

// Output:
// Function One
// Function One, Part 2
// (after a two-second delay)
// Function Two
```

- In functionTwo, instead of a normal console.log, we use setTimeout to simulate a **delay**, similar to fetching data from a server.
- The setTimeout function schedules 'Function Two' to be logged after a **2000-millisecond delay**.
- When you run the script, 'Function One' and 'Function One,Part 2' are logged **immediately**.
- After a **two-second delay**, 'Function Two' is logged.

Why doesn't JavaScript wait?

- JavaScript is designed to be **non-blocking**. If the engine waited for every time-consuming task, it could lead to **unresponsive applications**, especially if users interact with the webpage during these delays.
- This **non-blocking behavior** allows the JavaScript engine to continue executing other code while waiting for **asynchronous tasks** to complete.

What is asynchronous JavaScript?

Asynchronous JavaScript involves code where some tasks take **time** to complete. These tasks run in the **background** while the JavaScript engine continues executing other lines of code. When the result of the asynchronous task is **available**, it is then used in the program.

Key Takeaways

- **Synchronous JavaScript**: Executes code line by line, waiting for each task to complete before moving on.
- **Asynchronous JavaScript**: Allows tasks to run in the background, enabling the engine to continue executing other code. This is essential for handling tasks like network requests without freezing the application.

Understanding the difference between **synchronous** and **asynchronous** JavaScript is vital for building applications that are both **efficient** and **responsive**. As we progress, you'll learn more about handling asynchronous operations using **promises**, **async/await**, and other techniques.

Async/Await

In this lesson, you'll learn about async/await, a modern addition to JavaScript that simplifies working with promises. Async/await provides a more **intuitive** and **readable** way to handle asynchronous operations, making your code look and behave more like **synchronous** code.

Async/Await

Async/await is a syntactic sugar built on top of promises. It allows you to write asynchronous code that looks and behaves like synchronous code, which makes it easier to **read** and **maintain**.

Advantages

- **Readability**: Asynchronous functions using async/await resemble synchronous functions, making them easier to understand.
- **Error Handling**: You can use try/catch blocks to handle errors, similar to synchronous code.

Example

Let's take a look at a simple example:

```
const fetchNumber = async () => {
    return 25;
}

fetchNumber().then(result => {
    console.log(result); // should log 25
});
```

- The fetchNumber function is declared with the async keyword, which means it returns a **promise**.
- Inside the function, return 25; is equivalent to return Promise.resolve(25);.
- The .then() method is used to handle the resolved value of the promise.

Await

The await keyword is used to pause the execution of an async function until a promise is fulfilled. It can only be used inside an async function.

- **Non-blocking**: Using await does not block the entire program; it only pauses the execution of the async function until the promise is resolved.

Refactoring with Async/Await

Let's refactor our previous example with callbacks and promises using async/await:

```
const displayData = async () => {
    try {
        const user = await fetchUser('Adrian');
        const photos = await fetchUserPhotos(user);
        const detail = await fetchPhotoDetails(photos[0]);

        console.log(detail);
    } catch (error) {
        console.error(error);
    }
}
```

- The displayData function is declared with the async keyword, allowing the use of await within it.
- Each await expression pauses the function execution until the promise is resolved, making the code appear **synchronous**.
- A try/catch block is used to handle any **errors** that might occur during the asynchronous operations.

Key Takeaways

- **Async/Await**: Provides a **cleaner** and more **readable** way to work with promises.
- **Await**: Pauses the execution of an async function until a promise is resolved, without blocking the entire program.
- **Error Handling**: Use try/catch blocks to handle errors in a manner similar to synchronous code.

By using async/await, you can write asynchronous code that is easier to **read** and **maintain**, making it a powerful tool for modern JavaScript development.

▼ **Async JavaScript & Callbacks Part 1**

In this lesson, you'll learn about handling **asynchronous operations** in JavaScript, focusing on **data fetching**, **callback functions**, and how to simulate asynchronous behavior. These concepts are essential for building applications that interact with external data sources, such as **APIs**.

Simulating Asynchronous Data Fetching

Let's start with a real-world scenario: fetching data from an **API**. **APIs**, or Application Programming Interfaces, allow you to access data from various services. When fetching data, the time it takes can vary based on factors like **data size** and **network speed**. Unlike setTimeout, where you specify a delay, real data fetching times are **unpredictable**.

Consider the following example where we simulate **fetching a user** from a database:

```
const fetchUser = (username) => {
    setTimeout(() => {
        return { user: username };
    }, 2000); // Simulating a delay of 2000ms
}

const user = fetchUser('test');

console.log(user); // undefined
```

- The fetchUser function simulates a **delay** using setTimeout.
- It attempts to return a user object after **2 seconds**.
- However, console.log(user) outputs undefined because the function doesn't return the data **immediately**. The return statement inside setTimeout doesn't affect the outer function's return value.

Callback Functions

To handle **asynchronous operations** like data fetching, we can use **callback functions**. A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

Here's how you can modify the fetchUser function to use a callback:

```
const fetchUser = (username, callback) => {
    setTimeout(() => {
        console.log('Now we have the user');
        callback({ user: username });
    }, 2000);
}

fetchUser('test', (user) => {
    console.log(user);
});
```

- The fetchUser function now accepts a callback parameter.
- Inside setTimeout, once the simulated **delay** is over, the callback function is called with the user data.
- When fetchUser is called, we pass a callback function that logs the user data to the **console**.

Key Takeaways

- **Asynchronous Operations**: These operations, like data fetching, do not complete **immediately** and require handling mechanisms to manage their completion.
- **Callback Functions**: These are a fundamental way to handle asynchronous operations in JavaScript. They allow you to specify what should happen **once an asynchronous task is complete** .

In future lessons, we'll explore more advanced techniques for handling asynchronous operations, such as **promises** and **async/await**, which provide more **elegant** and **manageable** ways to work with asynchronous code. Understanding these concepts is crucial for building applications that efficiently interact with external data sources.

▼ **Async JavaScript & Callbacks Part 2**

In this lesson, you'll learn about the challenges of using **callback functions** in JavaScript, particularly when dealing with multiple asynchronous operations. This pattern, known as "**callback hell**," can make code difficult to read and maintain. We'll explore this issue and introduce the concept of **promises** as a solution.

Expanding Our Asynchronous Example

This was just a simple example, but now, let's add more **complexity** to it. Later on, we're going to exchange callback functions for both **Promises** and **Async/Await**. To complicate it, imagine we're working on a social media platform. Once the user profile is fetched, we want to fetch their **photos**. So let's create a function for that:

```
const fetchUser = (username, callback) => {
    setTimeout(() => {
        console.log('Now we have the user');
        callback(username);
    }, 2000);
```

```
}

const fetchUserPhotos = (username, callback) => {
  setTimeout(() => {
    console.log('Now we have the photos');
    callback(['photo1', 'photo2']);
  }, 2000); // Simulating a delay
}

const user = fetchUser('test', (username) => {
  console.log(username);

  fetchUserPhotos(username, (userPhotos) => {
    console.log(userPhotos);
  });
});
```

This is already getting **messy**. Let's add just one more function, and you'll easily notice the **problem**.

```
const fetchPhotoDetails = (photo, callback) => {
  setTimeout(() => {
    console.log('Now we have the photo details');
    callback('details...');
  }, 2000); // Simulating a delay
}

const user = fetchUser('test', (username) => {
  console.log(username);

  fetchUserPhotos(username, (userPhotos) => {
    console.log(userPhotos);

    fetchPhotoDetails(userPhotos[0], (details) => {
      console.log(details);
    });
  });
});
```

Problem: Callback Hell

As you can see, if we use **callbacks**, we get this weird structure that just keeps moving to the **right**. If we added a few more callbacks, this is how it would look:

```
const user = fetchUser('test', (username) => {
  fetchUserPhotos(username, (userPhotos) => {
    fetchPhotoDetails(userPhotos[0], (details) => {
      fetchPhotoDetails(userPhotos[0], (details) => {
        fetchPhotoDetails(userPhotos[0], (details) => {
          fetchPhotoDetails(userPhotos[0], (details) => {
            console.log(details);
          });
        });
      });
    });
```

```
    });
});
```

This is called **callback hell**. It becomes **unreadable**.

In this code, we can see on the left side, there's a triangle-like structure to the indentation, infamously known as "**callback hell**."

In short, every function gets an argument which is another function that is called with a parameter that is the response from the previous one.

You might find this sentence baffling, which describes the **CALLBACK HELL**.

As in our case, you can only imagine how many callback functions we will have to make to display several results. It's **difficult** to manage a lot of **callback functions**. Even if you wrote them yourself, you're going to have a hard time understanding them once you come back to the code after some time!

This pattern of coding at a large scale is **not maintainable**, is **confusing**, and also violates the **DRY (Don't Repeat Yourself)** principle, making it a **bad practice** to follow. To resolve this issue, JavaScript introduced the concept of **promises**. In the next lesson, you'll fully understand how promises work, and we'll refactor this code to use promises.

▼ **Promises**

In this lesson, you'll learn about promises in JavaScript, which provide a more readable and manageable way to handle **asynchronous operations** compared to **callback functions**. Promises help you avoid "**callback hell**" and make it easier to handle both successful and failed asynchronous operations.

Promises

In the last lecture, we witnessed the challenges of **callback hell**. Promises come to the rescue by offering a **cleaner** and more **structured** way to handle asynchronous tasks.

What are promises?

They are **objects** that represent the eventual **completion** (or **failure**) of an asynchronous operation and its resulting value. Promises can either return the successfully fetched data or an **error**.

Creating a Promise

Let's create a simple promise:

```
// Creating the promise
const promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        console.log('Got the user');
        resolve({ user: 'Adrian' });
    }, 2000);
});

// Getting the data from the promise
promise.then((user) => {
    console.log(user);
});
```

- A promise is created using the Promise constructor, which takes a function with resolve and reject parameters.
- Inside the promise, you perform your **asynchronous operation**. If successful, call resolve with the result.
- Use .then() to handle the resolved value of the promise.

Handling Errors with Promises

Promises also make it easy to handle **errors**. You can replace resolve with reject to simulate an error:

```
// Creating the promise
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    console.log('Got the user');
    // resolve({ user: 'Adrian' });
    reject('Error');
  }, 2000);
});

// Getting the data from the promise
// .then gives us the result of the resolve and .catch gives us the result of the reject
promise.then((user) => {
  console.log(user);
}).catch((error) => {
  console.log(error);
});
```

- Use .catch() to handle **errors** when the promise is rejected.

Refactoring Callback Hell with Promises

Let's refactor the previous callback example using **promises**:

```
console.log(1);

const fetchUser = (username) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('Now we have the user');
      resolve(username);
    }, 2000);
  });
}

const fetchUserPhotos = (username) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('Now we have the photos');
      resolve(['photo1', 'photo2']);
    }, 2000);
  });
}

const fetchPhotoDetails = (photo) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log('Now we have the photo details');
      resolve('details...');
    }, 2000);
  });
}
```

```
// Calling the functions with promises
fetchUser('Adrian')
    .then((user) => fetchUserPhotos(user))
    .then((photos) => fetchPhotoDetails(photos[0]))
    .then((detail) => console.log(detail))
    .catch((error) => console.log(error));

console.log(2);
```

- Each function returns a promise, allowing you to chain them using .then().
- This approach is much **cleaner** and **easier to read** compared to nested callbacks.
- The .catch() method at the end handles any **errors** that occur in the promise chain.

Key Takeaways

- **Promises** provide a more **readable** and **manageable** way to handle asynchronous operations.
- They allow you to **chain operations** and **handle errors** gracefully.
- Promises help avoid **callback hell**, making your code easier to **maintain**.

In the next lesson, you'll learn about async/await, a recent addition to JavaScript that builds on promises to make asynchronous code even more **intuitive**.

▼ **Closures:** Understanding how inner functions access outer function variables

What are Closures?

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives 1 you access to an outer function's scope from an inner function. 2

1. developer.mozilla.org

developer.mozilla.org

2. geektastic.com

geektastic.com

How Closures Work

1. **Inner Functions and Lexical Scope:** When a function is defined inside another function (an inner function), it has access to the outer function's variables and parameters. This is called lexical scoping.
2. **Preserving Outer Scope:** Even after the outer function has finished executing, the inner function retains access to the outer function's variables. This is because the inner function forms a closure, "closing over" the outer function's scope.
3. **Returning Inner Functions:** Closures are often created when an inner function is returned from an outer function.

Sample Code and Explanation

JavaScript

```
function outerFunction(outerVariable) {
  function innerFunction(innerVariable) {
    console.log("Outer variable:", outerVariable);
    console.log("Inner variable:", innerVariable);
  }
  return innerFunction; // Return the inner function
}

const myFunction = outerFunction("Hello"); // outerFunction execution finishes here.
myFunction("World"); // innerFunction is called, but it remembers outerVariable.
```

```
// Output:
// Outer variable: Hello
// Inner variable: World
```

Explanation:

- `outerFunction` takes `outerVariable` as a parameter.
- `innerFunction` is defined inside `outerFunction`.
- `innerFunction` has access to `outerVariable` because of lexical scoping.
- `outerFunction` returns `innerFunction`.
- When `myFunction` is called (which is the returned `innerFunction`), it still has access to `outerVariable` ("Hello") even though `outerFunction` has finished executing.

Practical Uses of Closures

1. **Data Privacy and Encapsulation:** Closures can be used to create private variables and functions, preventing them from being accessed or modified from outside.JavaScript

```
function createCounter() {
  let count = 0; // Private variable

  return {
    increment: function() {
      count++;
    },
    getCount: function() {
      return count;
    }
  };
}

const counter = createCounter();
counter.increment();
counter.increment();
console.log(counter.getCount()); // 2
// console.log(counter.count); // undefined, count is not directly accessible
```

2. **Event Handlers:** Closures are often used in event handlers to preserve the context of variables.JavaScript

```
function createButtonClickHandler(message) {
  return function() {
    console.log("Button clicked:", message);
  };
}

const button = document.querySelector("#myButton");
button.addEventListener("click", createButtonClickHandler("Button 1 clicked!"));
```

3. **Callbacks and Asynchronous Operations:** Closures can be used to maintain state across asynchronous operations.JavaScript

```
function delayedLog(message, delay) {
  setTimeout(function() {
    console.log(message);
```

```
  }, delay);
}

delayedLog("This message will appear after 2 seconds.", 2000);
```

4. **Module Pattern:** Closures are the foundation of the module pattern, which is used to create self-contained modules with private and public members.JavaScript

```
const myModule = (function() {
  let privateVariable = "I'm private";

  function privateFunction() {
    console.log(privateVariable);
  }

  return {
    publicMethod: function() {
      privateFunction();
    }
  };
})();

myModule.publicMethod(); // "I'm private"
// console.log(myModule.privateVariable); // undefined
```

Key Points

- Closures allow inner functions to retain access to outer function variables.
- Closures are created when an inner function is returned from an outer function.
- Closures enable data privacy, event handling, and other useful patterns.
- Closures are a fundamental concept in JavaScript and are essential for writing advanced code.

Understanding closures is crucial for writing efficient and maintainable JavaScript code.

In this lesson, you'll learn about closures in JavaScript, a powerful concept that allows functions to access variables from their outer scope even after the outer function has finished executing.

Closures

Closures are a fundamental concept in JavaScript that enable functions to retain access to their **lexical scope**, even when the function is executed **outside of that scope**. This is particularly useful when you have a function defined inside another function.

Here's a basic example to illustrate closures:

```
const outer = () => {
  const outerVar = 'Hello!';

  const inner = () => {
    const innerVar = 'Hi!';
    console.log(innerVar, outerVar);
  };

  return inner;
};
```

```
const innerFn = outer();
innerFn();
```

How Closures Work

Normally, when you exit a function, all its **variables** “disappear” because nothing needs them anymore. But if you declare a function inside another function, the **inner function** can still be called later and access the variables of the **outer function**. For this to work, the outer function’s variables need to “stick around.” JavaScript handles this by keeping the variables alive instead of forgetting them, creating what is known as a "closure."

In other words, a closure gives you access to an outer function’s **scope** from an inner function. In JavaScript, closures are created every time a function is created, at **function creation time**.

Example of a Closure

```
const init = () => {
  const hobby = 'Learning JavaScript'; // hobby is a local variable created by init

  const displayHobby = () => { // displayHobby() is the inner function, a closure
    console.log(hobby); // using variable declared in the parent function
  };

  displayHobby();
};

init();
```

- In this example, init() creates a local variable called hobby and a function called displayHobby().
- The displayHobby() function is an inner function that is defined inside init() and is only available within the body of the init() function.
- However, since inner functions have access to the variables of outer functions, displayHobby() can access the variable hobby declared in the parent function, init().

Returning a Closure

```
const init = () => {
  const hobby = 'Learning JavaScript'; // hobby is a local variable created by init

  const displayHobby = () => { // displayHobby() is the inner function, a closure
    console.log(hobby); // using variable declared in the parent function
  };

  return displayHobby;
};

const myFunc = init();
myFunc();
```

Running this code has the same effect as the previous example, but with a twist: the displayHobby() **inner function** is returned from the **outer function** before being executed.

At first glance, it may seem **unintuitive** that this code still works. In some programming languages, the **local variables** within a function exist only for the duration of that function's execution. However, in JavaScript, functions

form **closures**. A **closure** is the combination of a **function** and the **environment** within which that function was declared. This environment consists of any **local variables** that were **in-scope** at the time the closure was created.

- In this case, myFunc is a reference to the instance of the function displayHobby created when init is run.
- The instance of displayHobby maintains a reference to its **lexical environment**, within which the variable hobby exists.
- For this reason, when myFunc is invoked, the variable hobby remains available for use.

Closures are a powerful feature in JavaScript, enabling more **flexible** and **modular** code. Understanding closures will enhance your ability to write **efficient** and **effective** JavaScript programs.

▼ `this` **Keyword:** Context-dependent `this` (global, function, method, arrow functions)

The `this` keyword in JavaScript is a source of much confusion, but understanding its context-dependent nature is crucial. Let's break down how `this` behaves in different scenarios.

Understanding `this`

The value of `this` is determined by how a function is *called*, not where it's defined. This dynamic nature can lead to unexpected results.

1. `this` in the Global Context

- **Browser:** In a browser environment, `this` in the global scope refers to the `window` object.
- **Node.js:** In Node.js, `this` in the global scope refers to the `global` object.
- **Strict Mode:** In strict mode (`'use strict'`), `this` in the global scope is `undefined`.

JavaScript

```
// Browser:
console.log(this); // window

// Node.js:
// console.log(this); // global

'use strict';
console.log(this); // undefined
```

2. `this` in a Function Context

- **Default Binding:** When a regular function is called independently (not as a method), `this` defaults to the global object (or `undefined` in strict mode).

JavaScript

```
function myFunction() {
  console.log(this);
}

myFunction(); // Browser: window, Node.js: global, Strict mode: undefined

'use strict';
function strictFunction(){
  console.log(this);
}

strictFunction(); // undefined
```

3. `this` in a Method Context

- **Implicit Binding:** When a function is called as a method of an object, `this` refers to the object that the method is called on.

JavaScript

```
const myObject = {
  name: "My Object",
  myMethod: function() {
    console.log(this.name);
  }
};

myObject.myMethod(); // "My Object"
```

- **Nested Methods:** If you have nested functions within a method, `this` will not automatically refer to the outer object. You'll need to use techniques like `bind()`, `call()`, or `apply()`, or store `this` in a variable.

JavaScript

```
const myObject2 = {
  name: "Outer Object",
  myMethod: function() {
    function innerFunction() {
      console.log(this); // Window (in browser), global (Node.js), or undefined (strict mode)
    }
    innerFunction(); //Called as a regular function.

    const that = this; // Store 'this'
    function innerFunction2() {
       console.log(that); // myObject2
    }
    innerFunction2();

    const innerFunction3 = () =>{
       console.log(this); // myObject2, arrow function
    }
    innerFunction3();
  }
};

myObject2.myMethod();
```

4. `this` in Arrow Functions

- **Lexical Binding:** Arrow functions do not have their own `this` context. They inherit `this` from the surrounding (lexical) scope. This makes them particularly useful for callbacks and nested functions.

JavaScript

```
const myObject3 = {
  name: "Arrow Object",
  myMethod: function() {
    const arrowFunction = () => {
      console.log(this.name);
    };
    arrowFunction();
```

```
  }
};

myObject3.myMethod(); // "Arrow Object"
```

5. `bind()`**,** `call()`**, and** `apply()`

- These methods allow you to explicitly set the value of `this` when calling a function.
- `call()`**:** Calls a function with a specified `this` value and arguments passed individually.
- `apply()`**:** Calls a function with a specified `this` value and arguments passed as an array.
- `bind()`**:** Creates a new function with a specified `this` value.

JavaScript

```
function sayHello(greeting) {
  console.log(greeting + ", " + this.name);
}

const person = { name: "Alice" };

sayHello.call(person, "Hello"); // "Hello, Alice"
sayHello.apply(person, ["Hi"]); // "Hi, Alice"

const boundFunction = sayHello.bind(person);
boundFunction("Greetings"); // "Greetings, Alice"
```

Key Takeaways

- The value of `this` depends on how a function is called.
- Use `bind()`, `call()`, or `apply()` to explicitly set `this`.
- Arrow functions inherit `this` from their surrounding scope.
- Be mindful of `this` in nested functions and callbacks.
- `this` inside a constructor function refers to the newly created object.
- Use strict mode to catch potential `this` errors.

In this lesson, you'll learn about the this keyword in JavaScript, which is a fundamental concept for understanding how **functions** and **objects** interact in different **contexts**.

Introduction

The this keyword in JavaScript refers to the **object** that is executing the **current function**. It provides a way to access the **properties** and **methods** of the object within which the function is being executed. Understanding this is crucial for working with object-oriented programming in JavaScript.

Using it in a Function

Let's explore how this works with a simple example:

```
JAVASCRIPT

1function Sentence(words) {
2  this.words = words;
3  console.log(this);
4}
```

```
5
6const S = new Sentence("hello there, we are learning about the `this` keyword");
```

Explanation

1. **Function Definition**: We define a function Sentence that takes an input parameter words.
2. **Assigning Properties**: Inside the function, we assign this.words to the input parameter words. This means that the words property of the object being created will hold the value passed to the function.
3. **Logging this**: We use console.log(this) to print the current context of this. When the function is called with the new keyword, this refers to the new object being created.
4. **Creating an Instance**: We create a new instance of Sentence by calling new Sentence(...). This creates a new object, and this within the function refers to this new object.

What Happens When You Run the Code

- When you run the code, the Sentence function is executed in the context of a **new object**. The this keyword inside the function refers to **this** new object.
- The console.log(this) statement outputs the new object, showing its properties, including words which is set to the string "hello there, we are learning about the this keyword".

Key Points to Remember

- **Context Matters**: The value of this depends on how a function is called. In the context of a constructor function (like Sentence), this refers to the **new object** being created.
- **Global Context**: If a function is called without an object context (e.g., not as a method or constructor), this refers to the global object (or undefined in strict mode).
- **Method Context**: When a function is called as a method of an object, this refers to the object the method is called on.

Understanding the this keyword is essential for working with **objects** and **functions** in JavaScript, as it allows you to access and **manipulate** the **properties** of the object within which a function is executed.

▼ **"new" Keyword**

In this lesson, you'll learn about the new keyword in JavaScript, which is crucial for creating objects and understanding how JavaScript handles object-oriented programming.

Introduction

The new keyword in JavaScript is used to create a new object. At its core, the new keyword performs a simple yet powerful function: **it creates a new object**. Let's explore this with some code examples.

Creating an Object with new

When you use the new keyword, you create an **empty object**. Here's how you can **create an object** using the new keyword:

```
const person = new Object();
```

This line of code creates an **empty object** called person. It's equivalent to writing person =. You can **add properties** to this object just like any other object:

```
person.lastname = "John";
console.log(person.lastname); // prints "John"
```

You can also check the **type** of the person object:

```
console.log(typeof person); // prints "object"
```

Object Methods & Object() Constructor

The Object() function is a **built-in constructor** in JavaScript that allows you to **create objects**. You can also define your own constructor functions to create objects of a specific type.

Creating Custom Constructors

Here's how you can create a custom constructor function:

```
function Person(name, age, profession) {
  this.name = name;
  this.age = age;
  this.profession = profession;
}

const john = new Person("John", 23, "Teacher");
console.log(john.name); // prints "John"
```

In this example, we created a Person constructor **function** and used the new keyword to create an **instance** of Person.

NEW & THIS (Keywords)

The new keyword **binds** this to the **new object** being created. In the Person constructor function, this refers to the **new** Person object.

```
function Person(name, age, profession) {
  this.name = name;
  this.age = age;
  this.profession = profession;
}
```

Built-in Constructors and Methods

JavaScript provides several built-in constructors like **Date**, **Array**, **Number**, etc. When you use the **new** keyword with these constructors, you create objects with built-in methods.

```
const myDate = new Date('August 11, 2025');
console.log(myDate.getFullYear()); // prints 2025
```

These constructors provide methods that you can use on the objects they create. For example, arrays have methods like **pop**, **push**, **slice**, and **splice**.

Literal Syntax vs. new Keyword

JavaScript also provides literal syntax for creating objects and arrays, which is a shorthand for using the **new** keyword.

```
const names = ['wes', 'kait'];
console.log(typeof names); // prints "object"
console.log(names instanceof Array); // prints true
```

The literal syntax is a more concise way to create objects and arrays, but under the hood, they are still objects with methods.

Understanding the **new** keyword and how it interacts with constructors and the **this** keyword is essential for mastering object-oriented programming in JavaScript.

▼ **Values and references**

Value vs Reference Introduction

JavaScript differentiates data types into **two categories**:

- **Primitive values**: Number, String, Boolean, Null, Undefined, etc.
- **Complex values**: Objects, Arrays

Understanding how these types are **copied** and **manipulated** is key to avoiding unexpected behavior in your code.

Copying Primitive Values

When copying **primitive values**, JavaScript behaves as you might expect. The value is copied directly, and changes to one variable do not affect the other. Here are some examples:

Copying Numbers

```
let x = 1;
let y = x;

x = 2;

console.log(x); // 2
console.log(y); // 1
```

Copying Strings

```
let firstPerson = 'Mark';
let secondPerson = firstPerson;

firstPerson = 'Austin';

console.log(firstPerson); // Austin
console.log(secondPerson); // Mark
```

In both cases, the original value is **copied**, and changes to the original variable **do not affect** the copied variable.

Copying Complex Values

When copying **complex values**, such as **objects** and **arrays**, JavaScript behaves differently. Instead of copying the value, it copies a **reference** to the value. This means changes to one variable **can affect the other** .

Copying Arrays

```
const animals = ['dogs', 'cats'];
const otherAnimals = animals;

animals.push('llamas');

console.log(animals); // ['dogs', 'cats', 'llamas']
console.log(otherAnimals); // ['dogs', 'cats', 'llamas']
```

Copying Objects

```
const person = {
   firstName: 'Jon',
   lastName: 'Snow',
};

const otherPerson = person;

person.firstName = 'JOHNNY';

console.log(person); // { firstName: 'JOHNNY', lastName: 'Snow' }
console.log(otherPerson); // { firstName: 'JOHNNY', lastName: 'Snow' }
```

In both examples, changes to the original array or object **also affect** the copied variable. This is because both variables **reference the same** underlying data.

Conclusion

This behavior can be surprising if you're not expecting it, but it makes sense once you understand that **complex values are stored as references**. In the next lesson, we'll dive deeper into why this happens and how you can manage it effectively in your code. Understanding the **difference between value and reference** is essential for writing robust and predictable JavaScript programs.

Understanding Reference in JavaScript

In our previous lesson, we explored how **primitive values** are copied by **value**, meaning each variable gets its own copy of the data. However, when it comes to **non-primitive values** like objects and arrays, JavaScript uses **references** .

Revisiting the Example

Let's revisit the example from our last lesson:

```
const person = {
   firstName: 'Jon',
   lastName: 'Snow',
};

const otherPerson = person;

person.firstName = 'JOHNNY';

console.log(person); // { firstName: 'JOHNNY', lastName: 'Snow' }
console.log(otherPerson); // { firstName: 'JOHNNY', lastName: 'Snow' }
```

What Happened Here?

When a **variable** is assigned a **primitive value**, it simply **copies** that value. We saw this with numbers and strings. However, when a variable is assigned a **non-primitive value** (such as an object, array, or function), it is given a **reference** to that object’s location in memory.

In the example above, the variable otherPerson doesn’t actually contain the value firstName: 'Jon' lastName: 'Snow'.

Instead, it points to a **location in memory** where that value is **stored**.

```
const otherPerson = person;
```

When a **reference** type value is copied to another variable, like otherPerson, the object is copied by **reference** instead of **value**. In simple terms, person and otherPerson don't have their **own copy** of the value. They point to the same location in memory.

```
person.firstName = 'JOHNNY';

console.log(person); // { firstName: 'JOHNNY', lastName: 'Snow' }
console.log(otherPerson); // { firstName: 'JOHNNY', lastName: 'Snow' }
```

When a property is **modified** on person, the object in memory is **updated**, and as a result, otherPerson also reflects that change.

Equality Check

We can demonstrate this behavior with a simple **equality check**:

```
const person = { firstName: 'Jon' };
const otherPerson = { firstName: 'Jon' };

console.log(person === otherPerson); // FALSE
```

You might expect person === otherPerson to resolve to true because they look identical. However, they point to **two distinct objects** stored in **different** locations in memory.

Now, let's create a **copy** of the person object by copying the **reference**:

```
const anotherPerson = person;

console.log(person === anotherPerson); // TRUE
```

person and anotherPerson hold **references** to the **same** location in memory and are therefore considered **equal**.

Conclusion

We've learned that **primitive values** are copied by **value**, while **objects** and **arrays** are copied by **reference**. This means that changes to one reference affect all references pointing to the same object. In the next lesson, we'll explore how to make a **true copy** of an object, allowing you to **modify one without affecting the other**. Understanding these concepts is essential for writing robust and predictable JavaScript code.

▼ **Cloning**

Cloning Arrays

When you want to create a **copy** of an array **without keeping a reference** to the original, you can use several methods. Let's explore these methods and understand how they work.

- **Spread Operator**
- **Array.slice()**

Spread Operator ...

The **spread** operator ... is a modern addition to JavaScript that allows you to "spread" the **values** of **arrays**, **objects**, and **strings**. It's a concise way to **clone** arrays.

How to use

Imagine you have an array:

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

The spread operator ... takes all the elements from the numbers array and **spreads** them into a **new** array newNumbers.

```
console.log(...numbers); // 1, 2, 3, 4, 5
console.log(newNumbers); // [1, 2, 3, 4, 5]
```

Checking Equality

Let's compare the **original** and **copied** arrays:

```
const copiedNumbers = numbers;

console.log(numbers === copiedNumbers); // true
console.log(numbers === newNumbers); // false
```

- copiedNumbers points to the **same** memory location as numbers, so changes to one **affect** the other.
- newNumbers is a **separate** array, so changes to numbers **do not affect** it.

Modifying the Original Array

```
numbers.push(6);

console.log(numbers); // [1, 2, 3, 4, 5, 6]
console.log(copiedNumbers); // [1, 2, 3, 4, 5, 6]
console.log(newNumbers); // [1, 2, 3, 4, 5]
```

The newNumbers array remains **unchanged**, demonstrating that it is a "**shallow clone**".

Array.slice()

The slice() method can also be used to **clone** an array:

```
const numbers = [1, 2, 3, 4, 5];
const numbersCopy = numbers.slice();

console.log(numbersCopy); // [1, 2, 3, 4, 5]
```

Cloning Objects

Just like arrays, **objects** can be **cloned** using similar methods.

- **Spread Operator**
- **Object.assign()**

Spread Operator ...

You can **clone** objects using the spread operator:

```
const person = {
   name: 'Jon',
   age: 20,
};
```

```
const otherPerson = { ...person };

// Modify the cloned object
otherPerson.age = 21;

console.log(person); // { name: 'Jon', age: 20 }
console.log(otherPerson); // { name: 'Jon', age: 21 }
```

Object.assign()

The Object.assign() method can also be used to **clone** objects:

```
const A1 = { a: "2" };
const A2 = Object.assign({}, A1);

console.log(A2); // { a: "2" }
```

Conclusion

We've learned **two different ways** to clone both **objects** and **arrays**: using the **spread operator**... and using Array.slice() or Object.assign().

These methods create **shallow clones** meaning they copy the **top-level** properties but not **nested** objects.

In the next lesson, we'll explore the **difference** between **shallow** and **deep** clones and how to create a **deep clone**. Understanding these concepts is crucial for managing data effectively in JavaScript.

Deep Cloning

Deep cloning involves creating a **complete copy** of an object, including all nested objects, so that changes to the new object **do not affect the original**. This is crucial when working with complex data structures.

Cloning an Object with Nested Objects

Consider the following object:

```
const person = {
   firstName: 'Emma',
   car: {
      brand: 'BMW',
      color: 'blue',
      wheels: 4,
   }
};
```

1. Using the Spread Operator ...

We can use the spread operator ... to create a **shallow copy** of the object:

```
const newPerson = { ...person };
```

This **removes** the reference from the outer object, allowing us to change properties like firstName **without affecting** the original:

```
newPerson.firstName = 'Mia';
```

```
console.log(person); // { firstName: 'Emma', car: { brand: 'BMW', color: 'blue', wheels: 4 } }
console.log(newPerson); // { firstName: 'Mia', car: { brand: 'BMW', color: 'blue', wheels: 4 } }
```

However, if we change a property of the **nested** car object:

```
newPerson.car.color = 'red';

console.log(person); // { firstName: 'Emma', car: { brand: 'BMW', color: 'red', wheels: 4 } }
console.log(newPerson); // { firstName: 'Mia', car: { brand: 'BMW', color: 'red', wheels: 4 } }
```

Both objects are affected because the car object is still **referenced**.

2. Creating a Deep Clone

To create a **deep clone**, we need to remove references from all nested objects. One way to achieve this is by using JSON.stringify and JSON.parse.

```
const newPerson = JSON.parse(JSON.stringify(person));
```

This method **converts** the object to a **string** and then back to an **object**, effectively **removing all references**.

Testing the Deep Clone

Let's test the **deep clone** by modifying the new object:

```
newPerson.firstName = 'Mia';
newPerson.car.color = 'red';

console.log(person); // { firstName: 'Emma', car: { brand: 'BMW', color: 'blue', wheels: 4 } }
console.log(newPerson); // { firstName: 'Mia', car: { brand: 'BMW', color: 'red', wheels: 4 } }
```

The original person object remains **unchanged**, confirming that newPerson is a **deep clone**.

Conclusion

Deep cloning is a powerful technique for working with complex objects in JavaScript. By using JSON.stringify and JSON.parse, you can create **independent copies** of objects, ensuring that changes to one do not affect the other. Mastering this concept is crucial for **managing data** effectively in your applications. If you're still a bit confused, take your time to review this section and practice with different examples. Mastery takes time!

▼ **Spread Syntax**

In this lesson, you'll learn about the **spread syntax** in JavaScript, a powerful feature introduced in **ES6** that allows you to expand elements of an iterable (like an array or string) in places where multiple arguments or elements are expected. This feature is particularly useful for working with arrays and objects, making your code more **concise** and **readable**.

Understanding Spread Syntax

- The **spread syntax** ... allows you to **expand an iterable into individual elements**.
- It is the **opposite** of rest syntax, which **collects multiple elements into a single array**.
- Spread syntax can be used in **function calls**, **array literals**, and **object literals**.

Using Spread in Function Calls

Spread syntax can replace Function.prototype.apply() to pass array elements as arguments to a function.

```
function add(x, y, z) {
  return x + y + z;
```

```
}

const numbers = [1, 2, 3];

console.log(add(...numbers)); // Expected output: 6
```

Using Spread in Array Literals

Spread syntax allows you to **create new arrays** by expanding elements from existing arrays.

```
const parts = ["shoulders", "knees"];
const lyrics = ["head", ...parts, "and", "toes"];
console.log(lyrics); // ["head", "shoulders", "knees", "and", "toes"]
```

Copying and Concatenating Arrays

Spread syntax can be used to **make shallow copies** of arrays or **concatenate** multiple arrays.

```
const arr1 = [0, 1, 2];
const arr2 = [3, 4, 5];

const combined = [...arr1, ...arr2];
console.log(combined); // [0, 1, 2, 3, 4, 5]
```

Using Spread in Object Literals

Spread syntax can also be used to **copy** and **merge objects**.

```
const obj1 = { foo: "bar", x: 42 };
const obj2 = { bar: "baz", y: 13 };

const mergedObj = { ...obj1, ...obj2 };
console.log(mergedObj); // { foo: "bar", x: 42, bar: "baz", y: 13 }
```

Conclusion

The **spread syntax** is a versatile tool in JavaScript that **simplifies** working with **arrays** and **objects**. By using spread syntax, you can write more **concise** and **readable** code, whether you're **passing** arguments to functions, **creating** new arrays, or **merging** objects. Practice using spread syntax in a code sandbox or console to deepen your understanding and see its benefits in action.

▼ **Object & Array Destructuring**

In this lesson, you'll learn about destructuring assignment in JavaScript, a feature introduced in **ES6** that allows you to **unpack values** from **arrays** and properties from **objects** into distinct variables. This feature simplifies the process of working with complex data structures and enhances code **readability**.

Destructuring Arrays

Destructuring arrays allows you to **extract values** from an array and **assign** them to variables in a **single**, **concise statement**. This is particularly useful when you want to work with specific elements of an array without manually accessing each index.

Basic Array Destructuring

Instead of accessing each element **individually**, you can use destructuring to **extract values**:

```
let introduction = ["Hello", "I", "am", "Sarah"];
let [greeting, pronoun] = introduction;

console.log(greeting); // "Hello"
console.log(pronoun);  // "I"
```

Declaring Variables Before Assignment

You can declare variables **before assigning** them values from an array:

```
let greeting, pronoun;
[greeting, pronoun] = ["Hello", "I", "am", "Sarah"];

console.log(greeting); // "Hello"
console.log(pronoun);  // "I"
```

Skipping Items in an Array

You can **skip elements** in an array by using commas , :

```
let [greeting, , , name] = ["Hello", "I", "am", "Sarah"];
console.log(greeting); // "Hello"
console.log(name);     // "Sarah"
```

Assigning the Rest of an Array

You can use the rest syntax to **collect remaining elements** into an array:

```
let [greeting, ...intro] = ["Hello", "I", "am", "Sarah"];
console.log(greeting); // "Hello"
console.log(intro);    // ["I", "am", "Sarah"]
```

Using Default Values

Default values can be **assigned to variables** if the array element is undefined:

```
let [greeting = "Hi", name = "Sarah"] = ["Hello"];
console.log(greeting); // "Hello"
console.log(name);     // "Sarah"
```

Swapping Values Using Destructuring

Destructuring can be used to **swap values between variables**:

```
let a = 3;
let b = 6;

[a, b] = [b, a];

console.log(a); // 6
console.log(b); // 3
```

Destructuring Objects

Destructuring objects allows you to **extract properties from an object and assign them to variables**. This is useful for working with objects where you only need specific properties.

Basic Object Destructuring

```
let user = { name: "Alice", age: 25 };
let { name, age } = user;

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

Renaming Variables

You can **rename** variables while destructuring:

```
let { name: userName, age: userAge } = user;
console.log(userName); // "Alice"
console.log(userAge);  // 25
```

Using Default Values

You can **assign default values** to variables if the property does not exist:

```
let { name, isAdmin = false } = { name: "Alice" };
console.log(isAdmin); // false
```

Destructuring in Function Parameters

Destructuring can be used directly in function parameters to **extract values** from **objects** or **arrays** passed as **arguments**.

Example:

```
function displayUser({ name, age }) {
  console.log(`Name: ${name}, Age: ${age}`);
}

let user = { name: "Bob", age: 30 };
displayUser(user); // Name: Bob, Age: 30
```

Conclusion

Destructuring assignment is a versatile feature that simplifies the process of extracting values from **arrays** and **objects**. By using destructuring, you can write **cleaner**, more **concise** code, making it **easier** to work with **complex data structures**. Practice using destructuring in a code sandbox or console to deepen your understanding and see its benefits in action.

▼ **Clean Code**

In this lesson, you'll discover why clean code is crucial for the success and sustainability of software projects. Clean code is more than just functional code; it's about crafting code that is **easy** to **read**, **maintain**, and **extend**. Let's delve into the significance of clean code and explore strategies to achieve it.

Why Clean Code Matters

The Foundation of Software

In today's digital world, software is integral to every organization, and the source code is its core asset. **Clean code** ensures that this asset remains a **strength** rather than a **liability**. It dictates how applications function and perform, making it essential for **security**, **reliability**, and **maintainability**.

Advantages for Developers

- **Skill Development**: Writing clean code enhances a developer's ability to identify, understand, and fix issues efficiently. It promotes learning best practices and improves coding skills.
- **Increased Productivity**: Clean code minimizes the need for constant rework and lengthy feedback loops, leading to higher productivity and quicker delivery.
- **Positive Work Environment**: Developers can focus on creative projects instead of untangling messy code, resulting in greater job satisfaction and pride in their work.

Benefits for Organizations

- **Risk Mitigation**: Clean code reduces security vulnerabilities and unexpected downtimes, safeguarding the organization's reputation and cutting costs.
- **Reduced Technical Debt**: By prioritizing code quality from the outset, organizations can avoid the costly burden of technical debt.
- **Enhanced Development Speed**: Streamlined processes and clean code standards boost development speed and team efficiency.

Challenges in Prioritizing Clean Code

Despite its importance, **clean code** is often overlooked due to **time constraints**, **budget pressures**, and a lack of understanding of its long-term benefits. Many stakeholders prioritize quick delivery of **functional code**, overlooking the hidden costs of poorly written code.

Strategies for Achieving Clean Code

- **Keep It Simple**: Focus on solving one problem at a time and keep the code as straightforward as possible. This reduces complexity and makes the code easier to understand and maintain.
- **Embrace Test-Driven Development (TDD)**: Implement automated testing to ensure code quality. TDD helps catch issues early and makes refactoring safer and more efficient.
- **Regular Refactoring**: Allocate time for refactoring to prevent technical debt from accumulating. Regularly improving the codebase keeps it maintainable and scalable.
- **Adopt SOLID Principles**: Follow the SOLID principles to write clean, maintainable code. These principles provide a framework for designing robust software architectures.
- **Code Reviews and Pair Programming**: Conduct regular code reviews and practice pair programming to catch issues early and share knowledge among team members.

Long-Term Benefits of Clean Code

- **Lower Maintenance Costs**: Clean code is easier to maintain, reducing the time and money spent on fixing bugs and implementing new features.
- **Improved Readability**: A clean codebase is easier to read and understand, crucial for onboarding new developers and ensuring long-term project sustainability.
- **Better Performance and Scalability**: Well-structured code leads to better performance and makes it easier to scale applications as requirements change.

Conclusion

Investing in **clean code** is an **investment** in the **future** of your software projects. By prioritizing clean code, you can save **time**, **money**, and **effort** in the long run. It leads to **higher quality products**, happier developers, and

more **successful projects**. Remember, the effort you put into writing clean code today will pay off in the long run, making your projects more **scalable**, **maintainable**, and **enjoyable** to work on.

▼ **Naming Conventions**

In this lesson, you'll learn about the importance of naming conventions in JavaScript and how they contribute to writing clean, understandable, and maintainable code. Naming conventions are not just about aesthetics; they play a crucial role in enhancing code readability and consistency. Let's explore the key naming conventions in JavaScript and why they matter.

Benefits

Clarity and Readability

In JavaScript, **clear** and **descriptive** names help convey the **purpose** and **functionality** of variables, functions, and classes. This **clarity** is essential for both the original developer and anyone else who may work with the code in the future. Well-chosen names make it **easier** to **understand** what the code does at a glance.

Consistency Across the Codebase

Consistent naming conventions create a **uniform structure** across the codebase, making it easier to navigate and understand. When developers adhere to a **common set of naming rules**, it reduces confusion and helps maintain a **cohesive code structure**.

Facilitating Collaboration

In collaborative environments, **clear and consistent** naming conventions are essential. They ensure that all team members cana **esily understand** and contribute to the code, reducing the likelihood of **errors** and **miscommunication**.

Common Naming Conventions in JavaScript

- **camelCase**
  - **Usage**: camelCase is commonly used for naming variables and functions in JavaScript.
  - **Example**: totalPrice, calculateTotalPrice
  - **Description**: The first word is lowercase, and each subsequent word starts with an uppercase letter. This style improves readability by visually separating words.
- **PascalCase**
  - **Usage**: PascalCase is typically used for naming classes and constructor functions.
  - **Example**: UserAccount, OrderDetails
  - **Description**: Similar to camelCase, but the first letter is also capitalized. This convention helps distinguish classes from other types of identifiers.
- **UPPER_SNAKE_CASE**
  - **Usage**: UPPER_SNAKE_CASE is often used for naming constants.
  - **Example**: MAX_VALUE, API_ENDPOINT
  - **Description**: All letters are uppercase, and words are separated by underscores. This style indicates that the value is constant and should not change.
- **Descriptive Names**
  - **Usage**: Choose names that clearly describe the purpose or function of the variable, function, or class.
  - **Example**: Use calculateTotalPrice instead of calc for a function that calculates the total price.
  - **Description**: Avoid vague or generic names that don't convey meaning. Descriptive names improve code readability and understanding.

- **Avoid Abbreviations**
  - **Usage**: While abbreviations can save space, they often reduce clarity. Use full words unless the abbreviation is widely recognized.
  - **Example**: Use userId instead of uid.
  - **Description**: Full words provide more context and make the code easier to understand.

Long-Term Benefits of Good Naming Conventions

- **Reduced Maintenance Effort**: Clear and consistent names make it easier to maintain and update the code, reducing the time and effort required for future changes.
- **Improved Onboarding**: New developers can quickly understand and contribute to the codebase when naming conventions are followed, speeding up the onboarding process.
- **Enhanced Code Quality**: Good naming conventions contribute to overall code quality, making the codebase more robust and less prone to errors.

Conclusion

Naming conventions are a **fundamental** aspect of writing **clean** and **maintainable** JavaScript code. By adopting clear and consistent naming practices, you can **improve** the **readability** and **quality** of your code, facilitate collaboration, and reduce maintenance efforts. Remember, the effort you put into choosing the right names today **will pay off in the long run**, making your projects more **successful** and **enjoyable** to work on.

▼ **Organizing Code**

In this lesson, you'll discover how to structure your JavaScript code effectively by utilizing **modules** and other **best practices**. While JavaScript's **flexibility** allows for creative solutions, it can also lead to **disorganized code** if not managed properly. Let's explore strategies to keep your **codebase clean**, **scalable**, and **easy to maintain**.

Why Code Organization Matters

JavaScript's lack of inherent structure means that without a clear organizational strategy, your code can become chaotic. **Proper organization** is key for:

- **Readability:** A well-structured codebase is easier to read and understand, making it more accessible to other developers and your future self.
- **Maintainability:** A clear structure simplifies updates and bug fixes, reducing the risk of errors.
- **Scalability:** Organized code can be more easily extended and adapted as your application grows.

Harnessing JavaScript Modules for Better Structure

Modules in **JavaScript** allow you to encapsulate and organize code into **separate files**, making it easier to manage complex applications. They enable you to break your code into smaller, **reusable** parts with clear interfaces.

Creating a Module

In ES6, each JavaScript file acts as a module. You can **export** functions, variables, or classes using the export keyword:

/ math.js (module file)

```
// math.js (module file)
export function add(a, b) {
  return a + b;
}
```

/ main.js

```
// main.js
import { add } from './math.js';

const result = add(2, 3);
console.log(result); // 5
```

Default Exports

Modules can also have a single "**default**" export, ideal for exporting a primary function or class:

/ calculator.js (module file)

```
// calculator.js (module file)
export default function add(a, b) {
  return a + b;
}
```

/ main.js

```
// main.js
import add from './calculator.js';

const result = add(2, 3);
console.log(result); // 5
```

Advantages of Using Modules

- **Encapsulation**: Modules encapsulate functionality, reducing global variable conflicts and promoting a more organized code structure.
- **Reusability**: Modules can be reused across different parts of an application or in other projects.
- **Ease of Maintenance**: Modules allow for easier management and updates of individual code pieces without affecting the entire application.
- **Clear Dependency Management**: Modules help define dependencies clearly, making it easier to understand the relationships between different parts of your application.

Additional Tips for Organizing JavaScript Code

- **1. Comment Your Code**
- **Purpose**: Comments provide context and explain the reasoning behind your code, aiding understanding.
- **Best Practice**: Use comments to clarify complex logic and document assumptions. Keep them concise and relevant.
- **2. Utilize ES6 Classes**
- **Purpose**: Classes offer a clear structure for organizing business logic and promoting code reuse.
- **Best Practice**: Use classes to encapsulate related properties and methods, leveraging inheritance to avoid code duplication.
- **3. Embrace Promises**
- **Purpose**: Promises provide a cleaner way to handle asynchronous operations compared to callbacks.
- **Best Practice**: Use promises to chain asynchronous calls in a readable and organized manner, enhancing code clarity.
- **4. Separate Concerns**

- **Purpose**: Keeping different parts of your application separate enhances modularity and maintainability.
- **Best Practice**: Organize your code into modules, separating logic related to different features or components.
- **5. Define Constants and Enums**
- **Purpose**: Constants and enums help avoid hardcoding values, making your code more flexible and easier to update.
- **Best Practice**: Define constants for repeated values and use enums to group related constants.

Conclusion

Organizing your JavaScript code is crucial for building maintainable and scalable applications. By leveraging **modules**, commenting effectively, utilizing **ES6** features, and separating concerns, you can keep your codebase clean and efficient. The effort you invest in organizing your code today will pay off in the long run, making your projects more successful and enjoyable to work on.

▼ **Writing Small Functions**

In this lesson, you'll explore the benefits of writing small functions in **JavaScript** and how they contribute to clean, maintainable code. Small functions are a cornerstone of good software design, promoting **readability**, **reusability**, and ease of **testing**. Let's dive into why small functions matter and how to effectively implement them in your code.

Small Functions

Do One Thing (DOT) Principle

The **Do One Thing** principle, often referred to as **Curly's Law**, suggests that functions should perform **a single task**. This approach ensures that each function is **focused** and easy to understand. A function should either perform an action or return a result, but not both.

- **Example**: A function named updateUser should only handle updating user information, not fetching or saving it.

Consistency in Abstraction Levels

According to the **Same Level of Abstraction Principle (SLAP)**, all code within a function should operate at the same level of abstraction. This consistency makes the code easier to read and understand. Mixing different levels of abstraction within a single function can lead to confusion.

Guidelines for Writing Small Functions

- **Keep It Short and Sweet**
- **Screen Fit**: Aim for functions that fit on a single screen with a reasonable font size. This makes them easier to read and understand.
- **Three Blocks Rule**: Limit functions to about three logical blocks or "paragraphs" of code. Each block should represent a distinct step in the function's process.
- **Avoid Repetition**
- **DRY Principle**: Longer functions often contain repetitive code. By keeping functions small, you can eliminate redundancy and make the code more efficient.
- **Facilitate Testing and Debugging**
- **Unit Testing**: Smaller functions are easier to test individually, allowing for more precise debugging and validation of functionality.
- **Refactoring**: Small functions simplify the process of refactoring, making it easier to improve code quality over time.
- **Enhance Readability**
- **Readable Code**: Code that reads like a story is easier to maintain and extend. Small functions contribute to this by breaking down complex logic into manageable pieces.

Addressing Concerns About Small Functions

Some argue that **shorter functions** may lead to higher **defect density** or make **debugging** harder. However, with modern practices like **Test-Driven Development (TDD)** and powerful development tools, these concerns are mitigated. **Readable code** is inherently easier to maintain, and **small functions** support this goal.

When to Refactor

If writing **small functions** disrupts your workflow, start with a **longer function** and **refactor** it into smaller parts. Use your **IDE's refactoring tools** to extract methods and move them between **classes** until the code is **clear** and **concise**.

Conclusion

There is no strict rule for how many lines a function should have, but the goal is to keep functions as **short as possible** while maintaining **clarity**. **Small functions** enhance **readability**, facilitate **testing**, and reduce **complexity**, making your codebase more **robust** and **maintainable**. Remember, the effort you put into writing **small, focused functions** today will pay off in the long run, making your projects more **successful** and **enjoyable** to work on.

▼ **Avoiding Code Duplication**

In this lesson, you'll explore the DRY Principle—a fundamental concept in software development that stands for **"Don't Repeat Yourself."** Adhering to the DRY principle is crucial for maintaining a clean, efficient, and scalable codebase. Let's delve into why repetition can be detrimental to your code and how embracing DRY practices can benefit both developers and organizations.

Code Duplication

Technical Debt and Maintenance Challenges

Duplicating code across your **codebase** is akin to taking out a "**loan**" that accrues **interest** over time. Each instance of duplicated **code** becomes a potential point of **failure**, requiring separate **maintenance** and **updates**. This accumulation of **technical debt** can significantly slow down **development** and increase **project costs**.

- **Increased maintenance costs**: With duplicated code, any change must be replicated across multiple locations, multiplying your workload and increasing the risk of missing updates.
- **Bug propagation**: A bug in duplicated logic will exist in multiple places, leading to inconsistencies and potential errors if not addressed everywhere.

Reduced Readability and Scalability

Code duplication can lead to a **cluttered codebase**, making it harder to **read** and **understand**. This can hinder **scalability** and make it challenging to onboard new developers.

- **Complexity**: Repeated code adds unnecessary **complexity**, making it difficult to navigate and modify the codebase.
- **Inconsistencies**: Duplicated code can lead to **inconsistencies** in behavior, especially if changes are made in one place but not in others.

Embracing the DRY Principle

The **DRY principle** advocates for eliminating **duplication** in code by ensuring that every piece of **knowledge** has a single, unambiguous representation within a **system**. Here's how you can implement **DRY** practices:

- **Identify and Abstract Common Patterns**
- **Functions and Modules**: Encapsulate repeated logic into **functions** or **modules**. This not only reduces duplication but also promotes **reusability** and **modularity**.
- **Design Patterns**: Utilize **design patterns** that encourage code reuse and abstraction, such as the **Singleton** or **Factory** patterns.
- **Refactor Regularly**

- **Continuous Refactoring**: Make **refactoring** a regular part of your development process. As you identify duplication, refactor the code to eliminate it and improve overall quality.
- **Leverage Object-Oriented Principles**
- **Inheritance** and **Polymorphism**: Use **object-oriented principles** to share common behavior among related classes, reducing duplication and enhancing code flexibility.
- **Adopt Methodologies and Systems**
- **CSS and JavaScript Methodologies**: Use methodologies like **BEM** for **CSS** and **DRY** for **JavaScript** to optimize code arrangement and output.

Benefits of the DRY Principle

- **Improved Maintainability**: A **DRY** codebase is easier to maintain, reducing the time and effort required for updates and bug fixes.
- **Enhanced Performance**: Avoiding repetitive code reduces waste, leading to faster load times and improved performance.
- **Economic Efficiency**: An optimized, reusable codebase improves productivity and code quality, ultimately reducing development costs.

The **DRY** principle is a cornerstone of clean and maintainable code. By eliminating **duplication**, you can create a more **efficient**, **scalable**, and **sustainable** codebase. Remember, the effort you put into adhering to **DRY** practices today will pay off in the long run, making your projects more **successful** and enjoyable to work on.

▼ **Commenting and Documentation**

In this lesson, you'll explore the importance of both code comments and documentation in software development. While they serve different purposes, both are essential for maintaining a **readable** and **maintainable** codebase. Let's dive into how to effectively use **comments** and **documentation** to enhance your code.

Code Comments vs. Documentation

The Role of Code Comments

Code comments, often referred to as inline documentation, are brief notes added directly within the code. They serve to explain the "**why**" behind specific lines or blocks of code, providing **context** and **reasoning** that might not be immediately obvious.

- **Explain the 'Why'**: Comments should clarify the intent and purpose of the code, helping developers understand the reasoning behind it. For example, instead of saying "This function reverses a string," explain why it does so, like "This function reverses a string to ensure compatibility with a specific API requirement."
- **Clarify Complex Logic**: Use comments to break down complex or non-obvious code, making it easier to understand and maintain. This is particularly useful for explaining algorithms or intricate logic.
- **Short Shelf-Life**: Comments can become outdated if the code changes but the comments do not, leading to potential confusion. Regular updates are essential to keep them relevant.

The Role of High-Level Documentation

High-level documentation provides a broader overview of the codebase, explaining **flows**, **patterns**, and the overall **architecture**. It offers a larger context that **code comments** cannot provide..

- **Explain Flows and Patterns**: Documentation can describe how different parts of the codebase interact and the rationale behind design decisions. It provides a big-picture view that helps developers understand the system as a whole.
- **Longer-Lasting**: High-level documentation is often more stable and requires less frequent updates than inline comments. It serves as a valuable reference for understanding the overall structure and design of the codebase.

- **Centralized Location**: Documentation should be easily accessible, ideally stored alongside the codebase in a version-controlled environment. This ensures that it is always up-to-date and available to all team members.

Using Code Comments and Documentation Together

Both code comments and **documentation** are crucial for a well-documented codebase. They complement each other by providing detailed explanations at different levels of abstraction.

- **Code Comments**: Use them for immediate, line-specific explanations that clarify the purpose and logic of the code. They are ideal for providing context and reasoning for specific implementations.
- **High-Level Documentation**: Use it to provide a comprehensive overview of the codebase, including architecture, design patterns, and system interactions. It helps developers understand the broader context and how different components fit together.

Best Practices for Commenting and Documentation

- **Keep Comments Concise and Relevant**: Comments should be brief and focused, providing valuable information without overwhelming the reader. Use clear language and avoid unnecessary details.
- **Update Comments as Code Evolves**: Regularly review and update comments to reflect changes in the code. This ensures that they remain accurate and helpful.
- **Use Docstrings or Standardized Formats**: Adopt standardized formats like docstrings for documenting functions, classes, and modules. They provide clear descriptions and can be used to generate documentation automatically.
- **Organize Information by Function**: Structure documentation by the function it describes, rather than by information type, to make it easier for users to find the information they need.

Conclusion

Both code comments and **high-level documentation** are invaluable assets for development teams. They enhance **code readability**, facilitate **collaboration**, and support efficient **maintenance**. By embracing both, you can optimize your **codebase** and improve **productivity**. Remember, effective **documentation** is not just a "nice to have"—it's a critical component of successful software development. Embrace **continuous documentation** to supercharge your **code comments**, optimize your **codebase**, and support your **development team**.

▼ **Learning Frontend**

In this lesson, you'll explore the diverse landscape of **frontend development frameworks**, which are essential for building dynamic and interactive user interfaces. If you're passionate about creating visually appealing **web applications**, understanding these frameworks will greatly enhance your development skills.

Why Learn Frontend Frameworks?

Frontend frameworks simplify the process of building complex user interfaces by providing pre-built components and tools. They help you manage the structure and behavior of your web applications, making development more **efficient** and **scalable**.

Key Frontend Frameworks

React

React is one of the most popular **frontend libraries**, known for its **component-based architecture** and efficient rendering. It uses **JSX**, a syntax extension that allows you to write **HTML elements** in **JavaScript**, making it easy to integrate logic and UI.

- **Concurrent Mode**: React's Concurrent Mode enhances performance by allowing React to handle multiple tasks simultaneously, improving load speeds.
- **State Management**: React efficiently updates only the components that need to change, thanks to its state management system.
- **Props**: These allow you to pass data between components, enabling reusability and flexibility.

Vue.js

Vue.js is a progressive framework that is easy to integrate into projects. It is known for its **simplicity** and **flexibility**, making it a great choice for both small and large applications.

- **Reactive Data Binding**: Vue provides a simple and intuitive way to bind data to the DOM, automatically updating the view when the data changes.
- **Component System**: Like React, Vue uses a component-based architecture, allowing you to build encapsulated and reusable UI elements.
- **Developer Tools**: Vue offers robust developer tools that enhance the development experience.

Angular

Angular is a comprehensive framework developed by Google, designed for building **large-scale applications**. It provides a robust set of tools and features, including a powerful template system and dependency injection.

- **Two-Way Data Binding**: Angular's two-way data binding ensures that changes in the UI are immediately reflected in the model and vice versa.
- **TypeScript**: Angular is built with TypeScript, offering static typing and advanced features for large applications.
- **Performance Enhancements**: Angular focuses on improving performance and load times through various optimizations.

Svelte

Svelte is a relatively new framework that compiles components into highly efficient JavaScript code. It is known for its **simplicity** and **performance**, as it eliminates the need for a **virtual DOM**.

- **No Virtual DOM**: Svelte compiles your code to efficient JavaScript at build time, resulting in fast and lightweight applications.
- **Reactivity**: Svelte's reactivity model is straightforward, making it easy to manage state and update the UI.
- **Zero-Runtime Compilation**: Svelte offers zero-runtime compilation for rapid load times, enhancing performance.

Astro

Astro is a modern framework focused on delivering **fast**, content-focused websites. It allows you to use components from **different frameworks** together, making it versatile for various projects.

- **Islands Architecture**: Astro uses an "islands" architecture, where only the interactive parts of a page are hydrated, improving performance.
- **Framework Agnostic**: You can use components from React, Vue, Svelte, and more within an Astro project, offering great flexibility.
- **Server-Side Rendering**: Astro emphasizes server-side rendering, delivering static HTML by default while hydrating only interactive parts, enhancing performance.

Conclusion

Each of these frameworks offers unique features and benefits, catering to different project needs and developer preferences. By exploring **React, Vue, Angular, Svelte**, and **Astro**, you'll be equipped with a diverse set of tools to create engaging and efficient web applications. As you continue your journey in frontend development, experiment with these frameworks to find the ones that best suit your style and project requirements.

Later on, you should specialize in the one you prefer, as the **job market** typically looks for **specialists** in a specific **technology**. Specializing in a particular **framework** or **library** can make you more competitive and increase your opportunities in the **job market**. Focus on mastering the tools that align with your **interests** and **career goals**!

▼ **Learning Backend**

Got it! Here's the content with the bullet-tick lists added, while keeping the bolding exactly as you initially provided:

In this lesson, you'll explore a variety of modern **JavaScript runtimes** and **frameworks** that are essential for **backend** and **full-stack development**. Understanding these tools will enhance your ability to build **dynamic**, **scalable**, and **efficient web applications**.

Node.js

Node.js is a powerful JavaScript runtime that allows you to execute **JavaScript** on the **server side**, outside of the browser. It is built on **Chrome's V8 JavaScript engine** and is widely used for building **scalable network applications**.

- **JavaScript Everywhere**: Node.js enables you to use JavaScript for both frontend and backend development, creating a seamless development experience.
- **Non-Blocking I/O**: Its event-driven, non-blocking I/O model makes Node.js efficient and suitable for real-time applications like chat apps and online gaming.
- **Rich Ecosystem**: With access to a vast library of packages through NPM, Node.js simplifies the development process.

Bun.js

Bun.js is a modern JavaScript runtime that aims to be a fast and efficient alternative to **Node.js**, focusing on **performance** and **developer experience**.

- **Performance**: Bun.js is designed to be fast, with improvements in JavaScript execution speed and module loading.
- **Built-in Tools**: It includes built-in tools for bundling, transpiling, and testing, reducing the need for additional dependencies.
- **Compatibility**: Bun.js aims to be compatible with Node.js, making it easier to transition existing projects.

Deno.js

Deno is a secure JavaScript and **TypeScript** runtime created by Ryan Dahl, the original creator of **Node.js**. It addresses some of **Node.js's** shortcomings, such as **security** and **module management**.

- **Security**: Deno runs in a secure sandbox by default, requiring explicit permission for file, network, and environment access.
- **TypeScript Support**: It has built-in support for TypeScript, allowing developers to write modern, type-safe code without additional configuration.
- **Simplified Module System**: Deno uses a URL-based module system, eliminating the need for a centralized package manager like NPM.

Next.js: A Fullstack Framework Alternative

Next.js is a React-based framework that extends the capabilities of **React** to include **full-stack development** features. It allows you to build both **frontend** and **backend logic** within the same project.

- **Server-Side Rendering (SSR)**: Next.js improves performance and SEO by generating HTML on the server for each request.
- **Static Site Generation (SSG)**: It pre-renders pages at build time, offering a balance between dynamic content and performance.
- **API Routes**: You can create API endpoints directly within your Next.js project, enabling seamless integration of frontend and backend logic.
- **Automatic Code Splitting**: Next.js optimizes load times and performance by splitting your code into smaller bundles.

Conclusion

Each of these **environments** and **frameworks** offers unique features and benefits, catering to different project needs and developer preferences. **Node.js** remains a staple for **server-side JavaScript**, while **Bun.js** and **Deno.js** offer

modern alternatives with enhanced **performance** and **security features**. **Next.js** provides a comprehensive solution for **full-stack development** with **React**. By exploring these technologies, you'll be well-equipped to create **dynamic**, **scalable**, and **efficient web applications**.

III. Advanced (Deepening Knowledge)

▼ **Modules:** Import/Export (ES Modules), CommonJS (Node.js)

What are Modules?

Modules are a way to organize your code into separate files, making it more manageable, reusable, and maintainable. They allow you to:

- **Encapsulate code:** Keep related code together.
- **Avoid naming conflicts:** Prevent variables and functions from colliding.
- **Promote code reuse:** Share code across different parts of your application.

1. ES Modules (ECMAScript Modules):

- The modern standard for JavaScript and TypeScript.
- Supported by modern browsers and Node.js (with some configuration).
- Uses `import` and `export` keywords.

Exporting:

- **Named Exports:** Export individual variables, functions, or classes.TypeScript

```
// math.ts
export const PI = 3.14159;

export function add(a: number, b: number): number {
  return a + b;
}

export class Calculator {
  // ...
}
```

- **Default Exports:** Export a single value as the default.TypeScript

```
// message.ts
const message = "Hello, modules!";

export default message;
```

Importing:

- **Named Imports:** Import specific named exports.TypeScript

```
// main.ts
import { PI, add, Calculator } from "./math";

console.log(PI);
console.log(add(5, 10));
const calc = new Calculator();
```

- **Default Imports:** Import the default export.TypeScript

```
// main.ts
import message from "./message";

console.log(message);
```

- **Importing Everything:** Import all exports into a namespace object.TypeScript

```
// main.ts
import * as MathModule from "./math";

console.log(MathModule.PI);
console.log(MathModule.add(5, 10));
const calc = new MathModule.Calculator();
```

- **Renaming Imports/Exports:**TypeScript

```
// math.ts
export { add as sum };

// main.ts
import { sum as addition } from "./math";
```

Best Practices (ES Modules):

- Use named exports for most cases.
- Use default exports for single values (e.g., a main component or utility function).
- Keep module files focused on a single purpose.
- Use file extensions. (.js, .ts, .mjs, .mts)
- Ensure your `tsconfig.json` file is configured correctly to emit ES Modules.

2. CommonJS (Node.js):

- The traditional module system used in Node.js.
- Uses `require()` and `module.exports` (or `exports`).

Exporting:

- `module.exports`**:** Export a single value or an object containing multiple exports.JavaScript

```
// math.js
const PI = 3.14159;

function add(a, b) {
  return a + b;
}

module.exports = {
  PI,
  add,
};
```

- `exports`**:** Export individual properties of the `module.exports` object.JavaScript

```
// math.js
exports.PI = 3.14159;

exports.add = function (a, b) {
  return a + b;
};
```

Importing:

- `require()` **:** Import modules.JavaScript

```
// main.js
const math = require("./math");

console.log(math.PI);
console.log(math.add(5, 10));
```

Best Practices (CommonJS):

- Use `module.exports` for clarity, especially when exporting a single value or a complex object.
- Use `exports` for simpler cases when exporting multiple named properties.
- Avoid mixing CommonJS and ES Modules in the same project unless you're using Node.js with the `-experimental-modules` flag or using a bundler.

TypeScript and Modules:

- TypeScript supports both ES Modules and CommonJS.
- The `module` option in your `tsconfig.json` file controls the module system used during compilation.
- Modern projects should generally use ES Modules.

`tsconfig.json` **Configuration:**

- **ES Modules:**JSON

```
{
  "compilerOptions": {
    "module": "ESNext", // or "ES2020", "ES2022", etc.
    "moduleResolution": "NodeNext", // or "Node16"
    // ... other options
  }
}
```

- **CommonJS:**JSON

```
{
  "compilerOptions": {
    "module": "CommonJS",
    "moduleResolution": "Node",
    // ... other options
  }
}
```

Key Differences Summary:

- **Syntax:** ES Modules use `import` and `export`, while CommonJS uses `require()` and `module.exports`.

- **Standards:** ES Modules are the modern standard, while CommonJS is primarily used in Node.js.
- **Static vs. Dynamic:** ES Modules are statically analyzable (imports and exports are known at compile time), while CommonJS is dynamic (imports are resolved at runtime).
- **Browser Support:** ES Modules are natively supported in modern browsers, while CommonJS is not.
- **Node.js Support:** Node.js supports both, but ES Modules require configuration in older versions.

▼ **Error Handling:** `try...catch` blocks

Error handling is crucial for writing robust and reliable TypeScript applications. `try...catch` blocks are the fundamental mechanism for handling runtime errors. Let's break down how they work and best practices.

Purpose of `try...catch`:

- **Catching Runtime Errors:** To gracefully handle errors that occur during the execution of your code, preventing your application from crashing.
- **Providing Fallback Logic:** To execute alternative code when an error occurs.
- **Logging and Reporting Errors:** To record error information for debugging and analysis.

Syntax:

TypeScript

```
try {
  // Code that might throw an error
} catch (error) {
  // Code to handle the error
} finally {
  // Code that always executes (optional)
}
```

Explanation:

1. `try` **Block:**
   - The code within the `try` block is executed.
   - If an error occurs during execution, the control flow immediately jumps to the `catch` block.
2. `catch` **Block:**
   - The `catch` block is executed only if an error occurs in the `try` block.
   - The `error` parameter (usually named `error`, but you can name it anything) contains information about the error.
   - It is best practice to narrow the type of error to something more specific than any.
3. `finally` **Block (Optional):**
   - The `finally` block is always executed, regardless of whether an error occurred or not.
   - It's typically used for cleanup operations (e.g., closing files, releasing resources).

Example:

TypeScript

```
function divide(a: number, b: number): number {
  if (b === 0) {
    throw new Error("Division by zero is not allowed.");
  }
  return a / b;
}

try {
  let result = divide(10, 2);
  console.log("Result:", result);

  result = divide(5, 0); // This will throw an error
  console.log("This will not be reached.");
} catch (error) {
  if (error instanceof Error) {
    console.error("An error occurred:", error.message);
```

```
  } else {
    console.error("An unknown error occured", error);
  }
} finally {
  console.log("Finally block executed.");
}

console.log("Program continues after try...catch.");
```

Best Practices:

1. **Be Specific with Error Types:**
   - When possible, use specific error types (e.g., `TypeError`, `RangeError`, custom error classes) to handle different error scenarios.
   - When catching the error, use the instanceof operator to narrow the type of error.
2. **Handle Errors Appropriately:**
   - Don't just catch errors and ignore them.
   - Log errors, display user-friendly messages, or take other appropriate actions.
3. **Use `finally` for Cleanup:**
   - Use the `finally` block to ensure that cleanup operations are always performed.
4. **Avoid Excessive `try...catch` Blocks:**
   - Don't wrap every line of code in a `try...catch` block.
   - Use them strategically to handle potential error points.
5. **Throw Meaningful Errors:**
   - When throwing errors, provide clear and informative error messages.
6. **Custom Error Classes:**TypeScript
   - Create custom error classes to represent specific error conditions in your application. This makes error handling more organized and readable.

```
class MyCustomError extends Error {
  constructor(message: string) {
    super(message);
    this.name = "MyCustomError";
  }
}

try {
  throw new MyCustomError("Something went wrong.");
} catch (error) {
  if (error instanceof MyCustomError) {
    console.error("Custom error:", error.message);
  }
}
```

7. **Asynchronous Errors:**TypeScript
   - When working with asynchronous code (Promises, `async/await`), use `try...catch` blocks within `async` functions or `.catch()` methods on Promises.

```
async function fetchData() {
  try {
    const response = await fetch("https://example.com/data");
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    const data = await response.json();
    return data;
  } catch (error) {
    console.error("Error fetching data:", error);
    throw error; // Rethrow to propagate the error
  }
}
```

By following these best practices, you can write more robust and maintainable TypeScript code that gracefully handles runtime errors.

▼ **Regular Expressions:** Pattern matching, Searching, Replacing

Regular expressions (regex) are powerful tools for pattern matching, searching, and replacing text. TypeScript, being a superset of JavaScript, supports regular expressions natively.

1. Creating Regular Expressions:

- **Literal Notation:**TypeScript

```
let regex = /pattern/flags;
```

  - `pattern`: The regular expression pattern.
  - `flags`: Optional flags that modify the behavior of the regex (e.g., `i` for case-insensitive, `g` for global search).

- **Constructor Notation:**TypeScript

`let regex = new RegExp("pattern", "flags");`

2. Pattern Matching:

- `test()`**:** Checks if a string matches a pattern (returns `true` or `false`).TypeScript

```
let regex = /hello/i;
let text = "Hello world";

console.log(regex.test(text)); // Output: true
```

- `exec()`**:** Searches a string for a match and returns an array containing match information (or `null` if no match is found).TypeScript

```
let regex = /(\d{3})-(\d{3})-(\d{4})/;
let phone = "My phone number is 123-456-7890.";

let match = regex.exec(phone);

if (match) {
  console.log("Full match:", match[0]); // Output: 123-456-7890
  console.log("Area code:", match[1]); // Output: 123
  console.log("First part:", match[2]); // Output: 456
  console.log("Last part:", match[3]); // Output: 7890
}
```

3. Searching:

- `search()`**:** Returns the index of the first match in a string (or `1` if no match is found).TypeScript

```
let regex = /world/;
let text = "Hello world";

console.log(text.search(regex)); // Output: 6
```

- `match()`**:** Returns an array of matches in a string (or `null` if no match is found).TypeScript

```
let regex = /apple/gi;
let text = "I have an apple and an APPLe.";
```

```
let matches = text.match(regex);

if (matches) {
  console.log(matches); // Output: ["apple", "APPLe"]
}
```

4. Replacing:

- `replace()` **:** Replaces matches in a string with a new string or a function.TypeScript

```
let regex = /cat/gi;
let text = "The cat sat on the mat.";

let newText = text.replace(regex, "dog");
console.log(newText); // Output: The dog sat on the mat.

let regex2 = /(\w+), (\w+)/;
let text2 = "Doe, John";

let newText2 = text2.replace(regex2, "$2 $1"); // $1 and $2 are backreferences
console.log(newText2); // Output: John Doe

let regex3 = /\d+/g;
let text3 = "I have 3 apples and 5 oranges.";

let newText3 = text3.replace(regex3, (match) => (parseInt(match) * 2).toString());
console.log(newText3); // Output: I have 6 apples and 10 oranges.
```

5. Regex Flags:

- `i` **(case-insensitive):** Matches regardless of case.
- `g` **(global):** Matches all occurrences.
- `m` **(multiline):** Enables multiline mode (affects `^` and `$`).
- `s` **(dotall):** Allows `.` to match newline characters.
- `u` **(unicode):** Enables full Unicode support.
- `y` **(sticky):** Matches only from the current position in the string.

6. Regex Patterns:

- **Character Classes:**
  - `[abc]`: Matches any character in the set.
  - `[^abc]`: Matches any character not in the set.
  - `[a-z]`: Matches any character in the range.
  - `.`: Matches any character (except newline).
  - `\d`: Matches any digit.
  - `\w`: Matches any word character (alphanumeric and underscore).
  - `\s`: Matches any whitespace character.
- **Quantifiers:**
  - : Matches zero or more occurrences.
  - `+`: Matches one or more occurrences.

- `?`: Matches zero or one occurrence.
- `{n}`: Matches exactly `n` occurrences.
- `{n,}`: Matches `n` or more occurrences.
- `{n,m}`: Matches between `n` and `m` occurrences.

- **Anchors:**
  - `^`: Matches the beginning of a string.
  - `$`: Matches the end of a string.
  - `\b`: Matches a word boundary.
- **Grouping and Capturing:**
  - `(pattern)`: Groups a pattern and captures the match.
  - `(?:pattern)`: Groups a pattern but does not capture the match.

Best Practices:

- **Test Your Regex:** Use online regex testers to verify your patterns.
- **Keep It Simple:** Avoid overly complex regex patterns if possible.
- **Use Flags Appropriately:** Choose the correct flags for your needs.
- **Escape Special Characters:** Escape special characters (e.g., `\.`, `\*`, `\+`) with a backslash when you want to match them literally.
- **Use Raw Strings:** If you are using the constructor notation, using template literals or raw strings can help prevent confusion with backslashes.
- **Comment Complex Regex:** Add comments to explain complex regex patterns.
- **Performance:** Be mindful of performance implications, especially with large strings or complex patterns.

Sources and related content

github.com

github.com

gist.github.com

gist.github.com

▼ **Design Patterns:** Common solutions to recurring problems (e.g., Singleton, Factory, Observer)

Design patterns are reusable solutions to commonly occurring problems in software design. They provide a template for how to solve a problem that can be used in many different situations. Let's explore some fundamental design patterns: Singleton, Factory, and Observer.

1. Singleton Pattern

- **Purpose:** Ensures that a class has only one instance and provides a global point of access to it.
- **Use Cases:** Managing resources, configuration settings, or logging systems where only one instance is needed.
- **Implementation:**

JavaScript

```
const Singleton = (function() {
  let instance;

  function createInstance() {
    const object = new Object("I am the instance");
    return object;
```

```
  }

  return {
    getInstance: function() {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();

const instance1 = Singleton.getInstance();
const instance2 = Singleton.getInstance();

console.log(instance1 === instance2); // true
```

- **Explanation:** The `Singleton` is an immediately invoked function expression (IIFE). It creates a closure that stores the `instance` variable. The `getInstance` method checks if an instance exists and creates one if not, ensuring only one instance is ever created.

2. Factory Pattern

- **Purpose:** Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.
- **Use Cases:** When you need to create objects of different types based on certain conditions, but you want to hide the creation logic.
- **Implementation:**

JavaScript

```
function Car(options) {
 this.doors = options.doors || 4;
 this.state = options.state || "brand new";
 this.color = options.color || "silver";
}

function Truck(options) {
 this.doors = options.doors || 2;
 this.state = options.state || "brand new";
 this.color = options.color || "white";
}

function VehicleFactory() {}

VehicleFactory.prototype.vehicleClass = Car;

VehicleFactory.prototype.createVehicle = function(options) {
 if (options.vehicleType === "truck") {
   this.vehicleClass = Truck;
 } else {
   this.vehicleClass = Car;
 }
 return new this.vehicleClass(options);
};

const factory = new VehicleFactory();

const car = factory.createVehicle({ vehicleType: "car", doors: 4 });
const truck = factory.createVehicle({ vehicleType: "truck", doors: 2 });

console.log(car instanceof Car); // true
console.log(truck instanceof Truck); // true
```

- **Explanation:** The `VehicleFactory` creates either a `Car` or `Truck` object based on the `vehicleType` option. The creation logic is encapsulated within the factory, making it easy to add new vehicle types without modifying existing code.

3. Observer Pattern

- **Purpose:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Use Cases:** Event handling, model-view-controller (MVC) architecture, and real-time data updates.
- **Implementation:**

JavaScript

```
function Subject() {
 this.observers = [];
}

Subject.prototype.subscribe = function(observer) {
 this.observers.push(observer);
};

Subject.prototype.unsubscribe = function(observer) {
 this.observers = this.observers.filter(obs => obs !== observer);
};

Subject.prototype.notify = function(data) {
 this.observers.forEach(observer => observer.update(data));
};

function Observer(name) {
 this.name = name;
 this.update = function(data) {
  console.log(`${this.name} received: ${data}`);
 };
}

const subject = new Subject();
const observer1 = new Observer("Observer 1");
const observer2 = new Observer("Observer 2");

subject.subscribe(observer1);
subject.subscribe(observer2);

subject.notify("Hello, observers!");

subject.unsubscribe(observer2);

subject.notify("Another message!");
```

- **Explanation:** The `Subject` maintains a list of `observers`. When the subject's state changes, it calls the `update` method of each observer. This decouples the subject from its observers, allowing them to be added or removed independently.

Best Practices

- **Choose the Right Pattern:** Select a pattern that best fits the problem you're trying to solve.
- **Keep It Simple:** Avoid over-engineering your code. Use patterns only when they provide a clear benefit.
- **Understand the Trade-offs:** Each pattern has its own advantages and disadvantages.
- **Document Your Code:** Clearly document your use of design patterns to make your code easier to understand and maintain.
- **Favor Composition over Inheritance:** While inheritance can be useful, composition often leads to more flexible and maintainable code.
- **Use Design Patterns as Guidelines:** Design patterns are not rigid rules. Adapt them to fit the specific needs of your project.

Sources and related content

kogui.app

kogui.app

blog.devgenius.io

blog.devgenius.io

github.com

github.com

www.cnblogs.com

www.cnblogs.com

github.com

github.com

▼ **Testing:** Unit testing, Integration testing (using frameworks like Jest, Mocha)

Testing is a crucial part of software development, ensuring your code works as expected. Let's explore unit testing and integration testing, focusing on frameworks like Jest and Mocha.

1. Unit Testing

- **Purpose:** Tests individual units or components of your code in isolation.
- **Focus:** Verifies that each function or module performs its intended task correctly.
- **Benefits:**
  - Finds bugs early in the development process.
  - Improves code quality and maintainability.
  - Facilitates refactoring.

2. Integration Testing

- **Purpose:** Tests how different units or components of your code work together.
- **Focus:** Verifies that the interactions between modules or services are correct.
- **Benefits:**
  - Ensures that the application's components integrate seamlessly.
  - Detects issues related to dependencies and interfaces.
  - Validates the overall system behavior.

Testing Frameworks

- **Jest:**
  - A popular testing framework developed by Facebook.
  - Provides a complete testing solution with features like mocking, code coverage, and snapshot testing.
  - Easy to set up and use.
  - Excellent for React applications.
- **Mocha:**
  - A flexible and widely used testing framework.
  - Requires additional libraries for assertions (e.g., Chai) and mocking (e.g., Sinon).
  - Provides a clean and extensible API.
  - Suitable for various JavaScript environments.

Example using Jest

JavaScript

```
// math.js (Code to be tested)
function add(a, b) {
```

```
  return a + b;
}

function subtract(a, b) {
  return a - b;
}

module.exports = { add, subtract };

// math.test.js (Unit tests)
const { add, subtract } = require("./math");

describe("Math functions", () => {
  test("adds 1 + 2 to equal 3", () => {
    expect(add(1, 2)).toBe(3);
  });

  test("subtracts 5 - 3 to equal 2", () => {
    expect(subtract(5, 3)).toBe(2);
  });
});

//Example of an Integration Test
//api.js (simulated api)
async function fetchUserData(userID){
    if(userID === 1){
        return {name: "John", id: 1};
    } else {
        throw new Error("User not found");
    }
}
module.exports = {fetchUserData};

//user.js (code that uses the api)
const {fetchUserData} = require("./api");
async function getUserName(userID){
    const user = await fetchUserData(userID);
    return user.name;
}
module.exports = {getUserName};

//user.test.js (Integration test)
const {getUserName} = require("./user");
describe("User module Integration", ()=>{
    test("gets the username for a valid user", async ()=>{
        const name = await getUserName(1);
        expect(name).toBe("John");
    });
    test("throws an error for an invalid user", async ()=>{
        await expect(getUserName(2)).rejects.toThrow("User not found");
    });
});
```

Example using Mocha and Chai

JavaScript

```
 // math.js (Code to be tested)
function add(a, b) {
  return a + b;
}

function subtract(a, b) {
  return a - b;
}

module.exports = { add, subtract };

// math.test.js (Unit tests)
const { add, subtract } = require("./math");
const assert = require("chai").assert;

describe("Math functions", () => {
  it("adds 1 + 2 to equal 3", () => {
    assert.equal(add(1, 2), 3);
  });

  it("subtracts 5 - 3 to equal 2", () => {
    assert.equal(subtract(5, 3), 2);
  });
});

//api.js (simulated api)
async function fetchUserData(userID){
    if(userID === 1){
        return {name: "John", id: 1};
    } else {
        throw new Error("User not found");
    }
}
module.exports = {fetchUserData};

//user.js (code that uses the api)
const {fetchUserData} = require("./api");
async function getUserName(userID){
    const user = await fetchUserData(userID);
    return user.name;
}
module.exports = {getUserName};

//user.test.js (Integration test)
const {getUserName} = require("./user");
const expect = require("chai").expect;

describe("User module Integration", ()=>{
    it("gets the username for a valid user", async ()=>{
        const name = await getUserName(1);
        expect(name).to.equal("John");
    });
    it("throws an error for an invalid user", async ()=>{
        try{
            await getUserName(2);
            expect.fail("Should have thrown an error");
        } catch (error){
            expect(error.message).to.equal("User not found");
        }
    });
});
```

Best Practices for Testing

- **Write Tests Early:** Write tests alongside your code or even before (test-driven development - TDD).
- **Test Edge Cases:** Test boundary conditions and unexpected inputs.
- **Keep Tests Isolated:** Ensure unit tests don't depend on external factors.
- **Use Descriptive Names:** Make test names clear and concise.
- **Aim for High Coverage:** Strive for good code coverage, but don't obsess over 100%.
- **Automate Testing:** Integrate tests into your build process.

- **Mock External Dependencies:** For unit tests, mock external APIs or databases.
- **Use Assertions:** Use assertions to verify expected outcomes.
- **Organize Tests:** Group related tests using `describe` blocks.
- **Test Asynchronous Code:** Use `async/await` or promises in your tests.
- **Integration tests should test real world scenarios:** Try to make sure your integration tests mimic how a real user would interact with your application.
- **Use CI/CD:** Implement Continuous Integration/Continuous Deployment to run tests automatically.

▼ **Performance Optimization:** Techniques for writing efficient JavaScript code

Performance optimization in JavaScript involves techniques to make your code run faster and more efficiently. Here's a breakdown of key areas and best practices:

1. Minimize DOM Manipulation

- **Problem:** DOM operations are expensive due to browser reflows and repaints.
- **Solutions:**
  - **Batch DOM Changes:** Group multiple DOM changes and apply them at once.
  - **Use Document Fragments:** Create DOM elements in memory and append them to the document in a single operation.
  - **Minimize Reflows/Repaints:** Avoid repeatedly changing styles that trigger layout recalculations.
  - **Use `textContent` over `innerHTML` when possible:** `textContent` is faster when you're only changing text.

2. Optimize Loops

- **Problem:** Inefficient loops can slow down your code.
- **Solutions:**
  - **Cache Array Length:** Store the array length in a variable to avoid recalculating it on each iteration.
  - **Use `for...of` or `forEach()`:** For iterating over array elements, these are often more performant than traditional `for` loops.
  - **Avoid nested loops when possible:** Nested loops increase time complexity.
  - **Use while loops when possible:** while loops can be faster than for loops in some situations.

3. Reduce Function Calls

- **Problem:** Excessive function calls can add overhead.
- **Solutions:**
  - **Memoization:** Cache the results of expensive function calls and reuse them for the same inputs.
  - **Debouncing and Throttling:** Limit the number of times a function is called in response to events like scrolling or resizing.
  - **Avoid recursive functions when possible:** Recursive functions can use up a lot of memory.

4. Optimize Memory Usage

- **Problem:** Memory leaks and excessive memory usage can degrade performance.
- **Solutions:**
  - **Avoid Global Variables:** Global variables can lead to memory leaks.
  - **Release Unused Objects:** Set variables to `null` to release references to objects.
  - **Use WeakMaps and WeakSets:** To hold references to objects without preventing them from being garbage collected.

- **Clean up event listeners:** Remove event listeners when they are no longer needed.

5. Asynchronous Optimization

- **Problem:** Inefficient asynchronous operations can block the main thread.
- **Solutions:**
  - **Use Promises or Async/Await:** They provide better error handling and readability than callbacks.
  - **Use** `requestAnimationFrame()` **:** For animations to synchronize with the browser's refresh rate.
  - **Web Workers:** Use web workers to perform computationally intensive tasks in a separate thread.
  - **Optimize API calls:** Reduce the number of API calls, and ensure that your API returns only the needed data.

6. Code Optimization

- **Problem:** Inefficient code can slow down your application.
- **Solutions:**
  - **Use Strict Equality (===):** Avoid type coercion.
  - **Minimize Variable Lookups:** Store frequently used variables in local scope.
  - **Use bitwise operators when possible:** Bitwise operators are often faster than arithmetic operators.
  - **Optimize regular expressions:** Regular expressions can be slow if they are not written efficiently.
  - **Use code minification and bundling:** To reduce the size of your JavaScript files.

7. Browser-Specific Optimizations

- **Problem:** Different browsers have different performance characteristics.
- **Solutions:**
  - **Use Browser Developer Tools:** Profile your code to identify performance bottlenecks.
  - **Test on multiple browsers:** To ensure consistent performance.
  - **Use CSS animations and transitions:** They are often hardware-accelerated.
  - **Optimize images:** Use appropriate image formats and sizes.

8. Network Optimizations

- **Problem:** Network requests can be slow.
- **Solutions:**
  - **Use a Content Delivery Network (CDN):** To cache static assets.
  - **Enable Gzip compression:** To reduce the size of HTTP responses.
  - **Use HTTP/2 or HTTP/3:** For improved network performance.
  - **Cache API responses:** To reduce the number of network requests.
  - **Lazy load resources:** Load images and other resources only when they are needed.

Tools for Performance Optimization

- **Browser Developer Tools (Chrome DevTools, Firefox Developer Tools):** For profiling, network analysis, and memory analysis.
- **Lighthouse (Chrome extension or Node.js module):** For auditing web page performance.
- **WebPageTest:** For analyzing web page performance from different locations.
- **Jest (for code coverage):** To identify areas of your code that are not being tested.

Key Principles

- **Measure First:** Profile your code to identify performance bottlenecks before making changes.

- **Optimize Selectively:** Focus on optimizing the parts of your code that have the biggest impact on performance.
- **Test Thoroughly:** Ensure that your optimizations don't introduce new bugs.
- **Keep It Simple:** Avoid over-engineering your code.
- **Stay Updated:** Keep up with the latest browser and JavaScript performance best practices.

▼ **Security Best Practices:** Preventing common vulnerabilities (e.g., XSS, CSRF)

Security Best Practices: Preventing Common Vulnerabilities

1. Cross-Site Scripting (XSS)

Explanation: XSS attacks occur when an attacker injects malicious client-side scripts (usually JavaScript) into web pages viewed by other users. When a victim's browser executes these scripts, the attacker can steal cookies, session tokens, deface websites, or redirect users to malicious sites.

Types of XSS:

- **Stored XSS:** Malicious script is permanently stored on the target server (e.g., in a database, forum post, comment section).
- **Reflected XSS:** Malicious script is reflected off a web server onto the user's browser (e.g., via a search result, error message).
- **DOM-based XSS:** The vulnerability exists in client-side code rather than server-side code.

Prevention Best Practices:

- **Input Validation and Sanitization:**
  - **Always validate user input on the server-side.** Don't rely solely on client-side validation, as it can be bypassed.
  - **Sanitize input:** Remove or escape potentially malicious characters or tags from user-supplied data before storing or displaying it. Libraries like DOMPurify for client-side or validator.js/OWASP ESAPI for server-side can help.
- **Output Encoding/Escaping:**
  - This is the most critical defense against XSS. Before displaying any user-supplied data in the HTML, JavaScript, or CSS context, encode it appropriately.
  - **HTML Escaping:** Convert characters like `<`, `>`, `&`, `"`, `'` into their HTML entities (`<`, `>`, `&`, `&quot;`, `&#x27;` or `&apos;`).
  - **JavaScript Escaping:** When placing user data inside JavaScript, escape special characters to prevent code injection.
  - **URL Encoding:** When placing user data into URLs, encode it.
- **Content Security Policy (CSP):**
  - A powerful HTTP response header that tells the browser which dynamic resources (scripts, styles, images, etc.) are allowed to load and from which sources. This significantly mitigates XSS by preventing the execution of unauthorized scripts.
  - **Example CSP Header:**
    `Content-Security-Policy: default-src 'self'; script-src 'self' https://trusted.cdn.com; style-src 'self' 'unsafe-inline';`
- **Set `HttpOnly` Flag on Cookies:**
  - Mark sensitive cookies (like session IDs) with the `HttpOnly` flag. This prevents client-side JavaScript from accessing these cookies, even if XSS is successfully exploited.
- **Security Headers:** Utilize other relevant HTTP headers like `X-Content-Type-Options: nosniff` to prevent MIME-type sniffing.

Sample Code (Output Encoding in Node.js with Express and a templating engine like EJS):

JavaScript

```
// Server-side (Express.js example)
const express = require('express');
const app = express();
const escapeHtml = require('escape-html'); // A library for HTML escaping

app.get('/search', (req, res) => {
  const query = req.query.q || ''; // User input
  // BAD: res.send(`You searched for: ${query}`); // Vulnerable to XSS

  // GOOD: Escape the output
  const safeQuery = escapeHtml(query);
  res.send(`You searched for: ${safeQuery}`);
});

// Client-side (when injecting content into DOM)
const userInput = "<script>alert('XSS Attack!');</script>";
const el = document.getElementById('output');

// BAD: el.innerHTML = userInput; // Vulnerable to XSS
// GOOD: el.textContent = userInput; // Automatically escapes HTML
// Or, if you truly need HTML and trust the source:
// import DOMPurify from 'dompurify';
// el.innerHTML = DOMPurify.sanitize(userInput);
```

2. Cross-Site Request Forgery (CSRF)

Explanation: CSRF attacks trick authenticated users into submitting a malicious request to a web application. The attacker crafts a request (e.g., a hidden form, an image tag) that, when loaded by the victim's browser, performs an action on a trusted site where the victim is already authenticated. The attacker leverages the victim's browser automatically sending legitimate cookies with the malicious request.

Prevention Best Practices:

- **CSRF Tokens (Synchronizer Token Pattern):**
  - The most common and effective defense. The server generates a unique, unpredictable, and secret token for each user session.
  - This token is embedded in all state-changing forms and AJAX requests.
  - The server verifies this token upon receiving a request. If the token is missing or incorrect, the request is rejected.
  - **How it works:**
    1. User requests a page with a form.
    2. Server generates a unique CSRF token and embeds it in the form (e.g., a hidden input field) and stores it in the user's session.
    3. User submits the form.
    4. Server receives the request, compares the token from the request with the one stored in the session.
    5. If they match, the request is processed; otherwise, it's rejected.
- **SameSite Cookies:**
  - An attribute for cookies that controls when cookies are sent with cross-site requests.

- `SameSite=Lax` (default for many modern browsers): Cookies are sent with top-level navigations (GET requests) but not with cross-site POST requests. Offers good protection against common CSRF.
- `SameSite=Strict`: Cookies are only sent with same-site requests. Provides stronger protection but can affect user experience in some legitimate cross-site scenarios (e.g., clicking a link from an external site).
- `SameSite=None` with `Secure`: Cookies are sent with all requests, but only over HTTPS. Requires careful use with CSRF tokens.

- **Referer Header Check:**
  - Verify the `Referer` (or `Referrer`) header on sensitive requests to ensure the request originated from your own domain.
  - **Caution:** The `Referer` header can be stripped by browsers or proxies, so it shouldn't be the sole defense.
- **Custom Request Headers:**
  - For AJAX requests, you can add a custom request header (e.g., `X-Requested-With: XMLHttpRequest`). Browsers enforce the Same-Origin Policy for custom headers, so a cross-site attacker cannot forge this header.

Sample Code (CSRF Token in Node.js with Express and `csurf` library):

JavaScript

```
// Server-side (Express.js example with `csurf` and `cookie-parser`)
const express = require('express');
const cookieParser = require('cookie-parser');
const csrf = require('csurf');
const bodyParser = require('body-parser'); // To parse form data

const app = express();

app.use(bodyParser.urlencoded({ extended: false })); // For parsing x-www-form-urlencoded
app.use(cookieParser());
app.use(csrf({ cookie: true })); // Use CSRF protection with cookie storage

// Example route for a form
app.get('/form', (req, res) => {
  // Pass the CSRF token to the template
  res.send(`
    <form action="/process" method="POST">
      <input type="hidden" name="_csrf" value="${req.csrfToken()}">
      <label for="message">Message:</label>
      <input type="text" id="message" name="message">
      <button type="submit">Submit</button>
    </form>
  `);
});

// Example route to process the form
app.post('/process', (req, res) => {
  // If the CSRF token is invalid, `csrf()` middleware will throw an error
  // before this block is reached.
  console.log('Form data:', req.body.message);
  res.send('Form processed successfully!');
});

// Error handling for CSRF
```

```
app.use((err, req, res, next) => {
  if (err.code !== 'EBADCSRFTOKEN') return next(err);
  res.status(403).send('CSRF token invalid or missing!');
});

// Start the server
// app.listen(3000, () => console.log('Server running on port 3000'));
```

General Security Best Practices

- **Keep Dependencies Updated:** Regularly update all libraries, frameworks, and tools to their latest versions to patch known vulnerabilities.
- **Input Validation:** Always validate *all* user input on the server-side, not just for XSS/CSRF, but also for SQL Injection, Command Injection, etc.
- **Least Privilege:** Give users and processes only the permissions they need to perform their tasks.
- **HTTPS Everywhere:** Always use HTTPS to encrypt communication between the client and server.
- **Strong Authentication and Authorization:** Implement robust authentication mechanisms (e.g., strong passwords, multi-factor authentication) and proper authorization checks.
- **Error Handling:** Implement robust error handling that doesn't leak sensitive information (e.g., stack traces, database details) to the client.
- **Logging and Monitoring:** Log security-relevant events and monitor your application for suspicious activity.
- **Security Headers:** Implement other important HTTP security headers like `X-Frame-Options`, `Strict-Transport-Security`, `Permissions-Policy`.
- **Regular Security Audits and Penetration Testing:** Periodically have your application professionally audited for security vulnerabilities.

- **Working with APIs:** Fetching data from external sources (REST APIs, GraphQL)
- **Web Sockets:** Real-time communication between client and server

IV. Beyond (Specialized Areas)

- **Frameworks and Libraries:** React, Angular, Vue.js (front-end), Node.js, Express.js (back-end)
- **Version Control (Git):** Managing code changes
- **Build Tools:** Webpack, Babel (bundling, transpiling)
- **Deployment:** Deploying web applications
- **Server-Side JavaScript (Node.js):** Building back-end applications
- **Databases:** Working with databases (SQL, NoSQL)
- **TypeScript:** Adding static typing to JavaScript
- **WebAssembly:** Running code written in other languages in the browser

▼ **Old**

Variables

var - global.

let - inside code block

constant - inside code block , immutable after initialization

Data types

Numbers

Strings

Bolleans

Null

Undefined

Object

Symbol

Statically typed

- each variable type is already known at compile time

Dynamically typed

the variable can contain any data, the variable can be a string at one moment and can be an int.

1. **Variables & Data Types :** var, let, const.
2. **Primitive types:** string, number, boolean, null, undefined, symbol.
3. **Null vs Undefined (imp):**
4. a. `null:` is explicitly assigned to a variable to indicate that it is intentionally empty or has no value.**Type**: It is of type `object`. (This is a historical quirk in JavaScript.)**Usage**: It's typically used when you want to intentionally assign an empty value to a variable or object, indicating that something is intentionally missing.
5. b. `undefined`: means that a variable has been declared but has not been assigned a value, or a function doesn't return anything.**Type**: It is its own type, `undefined`.**Usage**: It's the default value for uninitialized variables or function arguments that are not passed. It also indicates that a function didn't return anything.
6. **Reference types**: object, array, function.
7. **Control Structures**: if, else, switch
8. **Loops:** for, while, do-while, for...of, for...in
9. **Functions:**a. Function declarations vs expressionsb. Arrow functionsc. Immediately Invoked Function Expression (IIFE)d. Higher-order functions (functions that take other functions as arguments or return them)e. Callback functions
10. **Promises :** It is a way to handle asynchronous operations.a. Promise.all()b. Promise.resolve()c. Promise.then()d. Promise.any()e. Promise.race()f. Promise.reject()

8. **Async/await :** Allows you to write asynchronous code in a more synchronous-looking manner.

9. **Callback Function :** A **callback** is a function that is passed as an argument to another function and is executed after the completion of that main function.

10. **Closures :** A closure in JavaScript is **a function that has access to variables in its parent scope**, even after the parent function has returned.

11. **Scope :**

a. Global vs local scope

b. Function scope, block scope (with let and const)

12. **Hoisting :**

a. Variable hoisting

b. Function hoisting

13. **Event loop and task queue (microtasks and macrotasks)**

14. **Execution Context** : An **execution context** is the environment in which the code is executed.

a. Global Execution Context (GEC)

b. Function Execution Context

15. **Scope Chain & Execution Contexts :**

The S**cope chain** is a crucial concept that determines how variables are looked up in different contexts when a function or block of code is executed

An E**xecution context** is an abstract concept that represents the environment in which the JavaScript code is evaluated and executed. Every time a function is invoked or a block of code is run, a new **execution context** is created.

16. **Memoisation:** It is a technique used to optimize functions by **caching** the results of expensive function calls and reusing those results when the same inputs occur again. This helps avoid redundant computations, improving performance in scenarios where a function is called repeatedly with the same arguments.

17. **Debouncing** : Limits the rate at which a function gets invoked. Helps avoid multiple function calls for events that trigger frequently, such as keystrokes or resize events.

18. **Throttling** : Ensures that a function is called at most once in a given period of time, no matter how often the event is triggered.

19. **Currying** : Why: Currying transforms a function that takes multiple arguments into a series of functions that each take one argument. This is useful for partially applying arguments.

Where to use: Functional programming, reusing functions with fixed arguments.

20. **setTimeout(), setInterval(), and clearTimeout() :**

a. **setTimeout()** : Executes a function after a specified delay (in milliseconds).

b. **setInterval() :** Executes a function repeatedly at a specified interval (in milliseconds).

c. **clearTimeout():** Cancels a previously scheduled `setTimeout()` operation.

21. **Template Literals:** Template literals, also known as template strings, are a feature in JavaScript that allow for easier string interpolation and multi-line strings. They are denoted by backticks (`) instead of single or double quotes.

22. **LocalStorage & SessionStorage:**

`localStorage` : Known for storing data persistently across browser sessions, remaining available even after the browser is closed.

`sessionStorage` : Known for storing data only for the duration of a single browser session, clearing when the tab or browser is closed.

23. **Regular Expressions (RegExp):** A **Regular Expression** (RegEx or RegExp) is a sequence of characters that defines a search pattern. RegEx is primarily used for string searching and manipulation, allowing you to search, match, and replace patterns in text.

24. **this Keyword :** `this` keyword refers to the **context** in which a function is executed. It is a special keyword that behaves differently depending on how a function is called.

1. In the global execution context (outside any function), `this` refers to the **global object** (`window` in browsers, `global` in Node.js).

2. In a regular function (not in strict mode), `this` refers to the **global object** (`window` in the browser).

3. When a function is called as a method of an object, `this` refers to the **object** the method is called on.

25. **OOPs in JavaScript:**

Classes In JavaScript

Classes and Objects in JavaScript

How to create a JavaScript class in ES6

this Keyword JavaScript

New Keyword in JavaScript

Object Constructor in JavaScript

Inheritance in JavaScript

Encapsulation in JavaScript

Static Methods In JavaScript

OOP in JavaScript

Getter and Setter in JavaScript

26. **Operators :**

a. Arithmetic Operators: `+`, `-`, `*`, `/`, `%` b. Comparison Operators: `==`, `===`, `!=`, `!==`, `>`, `<`, `>=`, `<=` c. Logical Operators: `&&`, `||`, `!` d. Assignment Operators: `=`, `+=`, `-=`, `*=`, `/=` e. Unary Operators: `++`, `--`, `typeof`, `delete` f. Ternary Operator (imp) : `condition ? expr1 : expr2`

27. **Break and Continue**

- `break` (exit the loop)
- `continue` (skip to the next iteration)

28. **Parameters and Arguments :**

Parameters are the **variables** defined in the **function declaration** (or function signature) that specify what kind of values the function expects to receive when it is called.

Arguments are the actual values passed to the function when it is called.

29. **Destructuring : Destructuring** is a syntax in JavaScript that allows you to unpack values from arrays or properties from objects into distinct variables. It simplifies the process of extracting multiple properties or elements from an object or array, making your code cleaner and more readable.

a. **Array destructuring**

b. **Object Destructuring**

30. **Rest and Spread Operator :** The **Rest** and **Spread** operators are both represented by `...` in JavaScript, but they serve different purposes depending on how they are used.

The **Rest** operator is used to collect multiple elements and bundle them into a single array or object. It's mainly used in function parameters to collect arguments, or in destructuring to collect remaining properties.

The **Spread** operator is used to unpack elements of an array or object into individual elements or properties. It allows you to expand or "spread" an iterable (array or object) into individual elements or properties.

31.**Event Delegation :** Using event listeners on parent elements to handle child element events

32. **Higher-Order Functions :** A H**igher-Order Function** is a function that either takes one or more functions as arguments or returns a function as its result.

34. **Anonymous functions :** An **anonymous function** is a function that does not have a name. These functions are typically defined inline and can be assigned to variables, passed as arguments, or used in other places where a function is required.

Key Characteristics:

- **No Name**: The function is defined without a name.
- **Often Used Inline**: Commonly used as callback functions or passed as arguments to higher-order functions.
- **Can be Assigned to Variables**: Can be assigned to variables or properties just like any other value.

35. **Lexical scoping :** The process of determining the scopes of the variables or functions during runtime is called **lexical scoping**.

How Lexical Scoping Works:

- When you define a function, it has access to variables that are within its scope (i.e., variables declared inside the function and variables from outer functions, including the global scope).
- If a function is nested inside another, the inner function can access variables from the outer function (this is called **closures**).

36. **Array Methods :**

- `push()` , `pop()` , `shift()` , `unshift()`
- `concat()` , `slice()` , `splice()`
- `map()` , `filter()` , `reduce()` , `forEach()`
- `find()` , `findIndex()`
- `sort()` , `reverse()`
- `join()` , `split()`
- `indexOf()` , `includes()` , `lastIndexOf()`

37. **Object Methods :**

- `Object.assign()` , `Object.create()` , `Object.keys()` , `Object.values()` , `Object.entries()` , `Object.hasOwn()` , `Object.freeze()` , `Object.seal()`

38. **Prototypes :**

- Prototype chain :
- Inheritance using prototypes

39. **Classes**

- Class syntax, constructors, methods
- Inheritance using `extends`
- `super()` and `super()` constructor

40. **call(), apply(), and bind() :** for controlling the context of this

41. **Event bubbling and capturing :**

Event Bubbling occurs when an event is triggered on an element, and the event then "bubbles up" from the target element to its ancestor elements in the DOM tree. In most cases, events bubble up by default unless you specifically prevent it

Event Capturing is the opposite of event bubbling. The event is first captured by the **root element** and then trickles down to the target element.

42. **Generators & Iterators :**

Why: Generators allow for lazy evaluation, meaning they yield values on demand rather than all at once. Useful for large data sets or infinite sequences.

Where to use: Implementing custom iterators, lazy evaluation of sequences.

43. **WeakMap and WeakSet :**

Why: Helps with memory management in JavaScript. WeakMap and WeakSet allow garbage collection of keys or values when there are no more references to them.

Where to use: Managing references to objects without preventing garbage collection. For example, caching DOM nodes where you don't want to create memory leaks.

44. **Polyfill :**

Why: Adds support for features that are not natively available in older browsers by providing code that mimics modern functionality.

Where to use: Ensuring compatibility with older browsers (e.g., older versions of Internet Explorer) for new JavaScript features like Promise, fetch, etc.

45. **Prototypal Inheritance :**

Why: JavaScript uses prototypes for inheritance, rather than the classical object-oriented inheritance. Understanding how the prototype chain works is key to understanding JavaScript's inheritance model.

Where to use: Building object hierarchies, adding methods to constructors.

46. **Cookies :** Storing and retrieving cookies in JavaScript

47. **Advanced Array Methods**

- **Array.prototype.find() :** Finding the first element in an array that matches a condition
- **Array.prototype.filter():** Filtering elements based on a condition
- **Array.prototype.reduce():** Reducing an array into a single value
- **Array.prototype.map():** Creating a new array by applying a function to each element
- **Array.prototype.sort():** Sorting arrays with custom sorting functions

48. **Design Patterns :**

- **Module Pattern:** Encapsulating code into modules
- **Singleton Pattern:** Ensuring a class has only one instance
- **Observer Pattern:** Notifying multiple objects when one object's state changes.
- **Factory Pattern:** Provides a way to instantiate objects while keeping the creation logic separate from the rest of the application.
- **Strategy Pattern :**Allows you to define a strategy (algorithm) for a particular operation and change it at runtime.
- **Decorator Pattern:** Dynamically adding behavior to an object without affecting its structure.

49. **Lazy Loading :** Delaying loading of content until it's needed.

50. **Working with JSON**

- **JSON Basics**
- JSON syntax, parsing with `JSON.parse()`, stringifying with `JSON.stringify()`
- **Working with APIs**
- Fetching data from an API using `fetch()`
- Handling API responses with Promises or Async/Await

51. **DOM Manipulation**

- **DOM Selection**
- `document.getElementById()`, `document.querySelector()`, `document.querySelectorAll()`
- **Event Handling**
- Event listeners: `addEventListener()`, `removeEventListener()`
- `event.target`, `event.preventDefault()`, `event.stopPropagation()`
- **Modifying DOM Elements**
- Changing text, HTML, attributes, styles
- Adding/removing elements dynamically (`createElement()`, `appendChild()`, `removeChild()`)
- **DOM Traversal**
- `parentNode`, `childNodes`, `nextSibling`, `previousSibling`

52. **Error Handling :**

- `try...catch...finally:` Handling errors in synchronous code

- **Custom Errors:** Creating custom error classes
- **Throwing Errors:** `throw` keyword for throwing errors manually

53. **String Methods :**

`charAt()` , `charCodeAt()` , `concat()` , `includes()` , `indexOf()` , `lastIndexOf()` , `slice()` , `split()` , `toLowerCase()` , `toUpperCase()` , `trim()` , `replace()` , `search()` , `ma`

54. **Date methods :**

`Date.now()` , `Date.parse()` , `Date.UTC()` , `getDate()` , `getDay()` , `getFullYear()` , `getHours()` , `getMilliseconds()` , `getMinutes()` , `getMonth()` , `getSeconds()` , `getT`

55. **Generator:** A **generator** in JavaScript is a special type of function that allows you to pause and resume its execution.

`function*` , `yield` , `next()` , `return()` , `throw()` .

56. **JavaScript Proxy :** A **Proxy** in JavaScript is a special object that allows you to intercept and customize operations on objects, such as property access, assignment, function calls, and more. It acts as a wrapper for another object and can redefine fundamental operations (like `get` , `set` , `deleteProperty` , etc.) on that object.

Commonly Used Traps (Methods):

1. `get(target, prop, receiver)` : Intercepts property access.
2. `set(target, prop, value, receiver)` : Intercepts property assignment.
3. `has(target, prop)` : Intercepts the `in` operator.
4. `deleteProperty(target, prop)` : Intercepts property deletion.
5. `apply(target, thisArg, argumentsList)` : Intercepts function calls.
6. `construct(target, args)` : Intercepts the `new` operator.
7. `defineProperty(target, prop, descriptor)` : Intercepts property definition.

57. **Javascript Array and Object Cloning** : Shallow or Deep?

A **shallow clone** of an object or array creates a **new instance**, but it only copies the **top-level properties** or elements. If the original object or array contains references to other objects (nested objects or arrays), these inner objects are **not copied**. Instead, the shallow clone will reference the same objects.

A **deep clone** creates a completely independent copy of the original object or array. It recursively copies all the properties or elements, including nested objects or arrays. This means that deep cloning ensures that no references to nested objects are shared between the original and the clone.

58. **loose equality (==) and strict equality (===) :**

Loose equality compares two values for equality **after performing type coercion**. This means that the values are converted to a common type (if they are of different types) before making the comparison.

When using `==` , JavaScript attempts to convert the operands to the same type before comparing them.

Strict equality compares two values without performing any type conversion. It checks both the **value** and the **type** of the operands.

For `===` , the operands must be of the same type and value to be considered equal.

59. **Call by Value Vs Call by Reference :**

Call by Value : When an argument is passed to a function by value, a **copy** of the actual value is passed. Any changes made to the argument inside the function do not affect the original variable outside the function.

When it happens: This occurs when **primitive types** (like numbers, strings, booleans, `null` , `undefined` , and `symbols`) are passed to a function.

Call by Reference : When an argument is passed by reference, the **reference** (or memory address) of the actual object is passed to the function. This means any changes made to the argument inside the function will directly modify the original object outside the function.

When it happens: This occurs when **non-primitive types** (like objects, arrays, and functions) are passed to a function.

60. **JavaScript Set :** A **Set** in JavaScript is a built-in collection object that allows you to store unique values of any type, whether primitive or object references.

Key Characteristics of a Set:

1. **Unique Elements**: A Set automatically ensures that each value it contains is unique. If you try to add a duplicate value, it will be ignored.
2. **Ordered**: The elements in a Set are ordered, meaning the values are stored in the order they were added. However, Sets do not allow duplicate entries.
3. **Iterable**: Sets are iterable, so you can loop over the elements in a Set using `for...of`, or methods like `.forEach()`.
4. **No Indexes**: Unlike Arrays, Set elements are not accessed by an index. They are stored by insertion order, but you can't reference them by a number.

Basic Methods of a Set :

1. `add(value)` : Adds a value to the Set. If the value already exists, it does nothing (no duplicates).
2. `has(value)` : Checks if the Set contains the specified value. Returns `true` or `false`.
3. `delete(value)` : Removes the specified value from the Set.
4. `clear()` : Removes all elements from the Set.
5. `size` : Returns the number of elements in the Set.
6. `forEach(callback)` : Executes a provided function once for each value in the Set.

61. **JavaScript Map :** A **Map** in JavaScript is a built-in object that stores key-value pairs. Unlike regular JavaScript objects, **Maps** allow keys of any data type (including objects, functions, and primitive types like strings and numbers) to be used.

Maps also maintain the insertion order of their keys, making them useful in scenarios where order matters.

Basic Methods of a Map :

1. `set(key, value)` : Adds or updates an element with the specified key and value in the Map.
2. `get(key)` : Retrieves the value associated with the specified key.
3. `has(key)` : Checks if a Map contains a key.
4. `delete(key)` : Removes the element associated with the specified key.
5. `clear()` : Removes all elements from the Map.
6. `size` : Returns the number of key-value pairs in the Map.
7. `forEach(callback)` : Executes a provided function once for each key-value pair in the Map.
8. `keys()` : Returns an iterator object containing all the keys in the Map.
9. `values()` : Returns an iterator object containing all the values in the Map.
10. `entries()` : Returns an iterator object containing an array of `[key, value]` pairs.

62. **The Fetch API:** The Fetch API allows us to make async requests to web servers from the browser. It returns a promise every time a request is made which is then further used to retrieve the response of the request.

63. **Import/Export:**

Modules: In JavaScript, a module is a file that contains code you want to reuse. Instead of having everything in one file, you can split your code into separate files and then import what you need. This keeps the code clean, organized, and maintainable.

- **Imports**: This is how you bring in functionality from other modules into your current file.
- **Exports**: This is how you make variables, functions, classes, or objects from one module available for use in other modules.

64. **Pure Functions, Side Effects, State Mutation and Event Propagation:**

65. **Recursion:Recursion** is a fundamental programming concept where a function calls itself in order to solve a problem. Recursion is often used when a problem can be broken down into smaller, similar sub-problems. In JavaScript, recursion is useful for tasks like traversing trees, solving puzzles, and more.

Key Concepts:

1. **Base Case**: The condition that stops the recursion. Without a base case, recursion can lead to infinite function calls, causing a stack overflow error.
2. **Recursive Case**: The part of the recursion where the function calls itself with a smaller or simpler version of the problem.

66. **The apply, call, and bind methods:**

67. **window methods:**

`alert()`, `confirm()`, `prompt()`, `setTimeout()`, `setInterval()`, `clearTimeout()`, `clearInterval()`, `open()`, `close()`, `requestAnimationFrame()`.

68. **Mouse events:**

`click`, `dblclick`, `mousedown`, `mouseup`, `mousemove`, `mouseover`, `mouseout`, `mouseenter`, `mouseleave`, `contextmenu`.

69. **Keyboard events:**

`keydown`, `keypress`, `keyup`.

70. **Form events:**

`submit`, `change`, `focus`, `blur`, `input`, `reset`, `select`, `keypress`, `keydown`, `keyup`.

71. **Debugging:**

72. **Cross-Origin Resource Sharing (CORS)**:

73. **Web Workers**: A mechanism for running scripts in background threads, allowing JavaScript to perform computationally expensive tasks without blocking the main thread.

74. **Service Workers**: A script that runs in the background of your browser, enabling features like push notifications, background sync, and caching for offline functionality.

75. **Lazy Loading** or **Infinite Scrolling**) :

Lazy Loading and **Infinite Scrolling** are two techniques commonly used to enhance performance and user experience in web applications, especially when dealing with large amounts of data or media (like images, lists, or articles).

Lazy Loading is a design pattern in web development where resources (such as images, scripts, videos, or even content) are loaded only when they are required.

The main goal of lazy loading is to improve the initial loading time of a webpage by reducing the number of resources loaded initially.

Infinite Scrolling is a technique that automatically loads more content as the user scrolls down the page, typically without the need for pagination. This is widely used in social media platforms, news sites, and any web application that needs to display large datasets (e.g., Instagram, Twitter, Facebook).

76: **Progressive Web Apps (PWAs)**: Building web applications that work offline, provide push notifications, and have native-like performance (through service workers and other browser APIs).

77. **Server-sent events :** Server-sent events (SSE) are a simple and efficient technology for enabling **real-time updates from the server to the client** over a **single HTTP connection.**

78.**Strict Mode :** Strict Mode is a feature in JavaScript that ensures that you avoid errors and problematic features.

79. **Security:** (Not a JavaScript concept, but important to know)

- **Cross-Site Scripting (XSS)**
- **Cross-Site Request Forgery (CSRF)**
- **Content Security Policy (CSP)**

- **CORS (Cross-Origin Resource Sharing)**
- **JWT (JSON Web Tokens)**

80. **Temporal Dead Zone (TDZ) :** It is a term that refers to the period of time between the creation of a variable and its initialization in the execution context. During this time, the variable exists but cannot be accessed — attempting to do so will result in a **ReferenceError**.

The TDZ occurs for variables declared using `let` and `const` but **not for** `var` because `var` declarations are hoisted and initialized with `undefined`.

▼ **JavaScript Resources**

In this section, you'll discover a variety of resources and opportunities to help you practice and enhance your **JavaScript** skills. Whether you're a **beginner** or looking to deepen your understanding, these resources offer a range of exercises, projects, and **community support** to challenge and inspire you.

JavaScript Resources

Practicing **JavaScript** through exercises and projects is a great way to solidify your understanding and improve your coding skills. Here are some excellent platforms and opportunities to explore:

1. **Exercism** offers a wide range of coding exercises across multiple programming languages, including **JavaScript**. It provides a unique opportunity to receive feedback from **mentors**, helping you improve your code and learn best practices.
2. **CodeWars** is a community-driven platform where you can solve coding challenges, known as "**katas**," to improve your skills. The challenges range from **beginner** to **advanced** levels, allowing you to progress at your own pace.
3. **FreeCodeCamp** offers a comprehensive curriculum that includes **JavaScript** exercises, **projects**, and **certifications**. It's a great resource for structured learning and building a portfolio of projects.
4. **JavaScript30** is a free **30-day challenge** that focuses on building **30 projects** in 30 days using **vanilla JavaScript**. It's an excellent way to practice your skills and learn by doing, without relying on **libraries** or **frameworks**.

Join Our Community

We invite you to join our **community**, where you can connect with fellow learners and developers, participate in **workshops**, and collaborate on projects. Here's how you can get involved:

- **Discord Server**: Join our Discord community to engage in discussions, ask questions, and share your progress. It's a great place to connect with others who are also learning JavaScript. Join our Discord
- **Workshops and Webinars**: Participate in our regular workshops and webinars to gain insights from experienced developers and work on real-world projects.
- **Build Projects on Our Platform**: Practice by building projects on our platform, where you can apply what you've learned and receive feedback from peers and mentors.

Additional Tips for Practicing JavaScript

- **Set Goals**: Define what you want to achieve with your practice sessions, whether it's mastering a specific concept or building a particular type of project.
- **Consistency is Key**: Try to practice regularly, even if it's just for a short period each day. Consistency will help reinforce your learning.
- **Engage with the Community**: Sharing knowledge and experiences can be incredibly beneficial. Participate in discussions and collaborate on projects with others.

Conclusion

By utilizing these resources and joining our **community**, you'll have the support and opportunities you need to become proficient in **JavaScript**. Remember, the key to mastering any skill is **practice** and **persistence**. We look forward to seeing you in our community and watching you grow as a developer.