

JavaScript

Explanation with Examples

Design and Written by
Vijay Mulsania

C-145, IT Park,
Digital Vally,
Mota Varachha Surat.

www.vijaytutor.com



JavaScript

Explanation and Examples

Vijay Mulsaniya

Contents

- Basics Of JavaScript 1
- Control Structures 2
- Conditional Structures 8
- JavaScript Functions 13
- JavaScript Variables 19
- JavaScript Arrays 23
- JavaScript Objects 29
- JavaScript Events 33
- JavaScript String Methods 42
- Regular Expressions (regex) 46
- JavaScript Dates 51
- JavaScript Math Object 55
- Error handling 60
- JavaScript Class 64
- JavaScript Async 68
- JavaScript Best Practices 72
- Interview Questions and Answers 84

Chapter 1

Basics Of JavaScript

Welcome to the world of JavaScript!

Let's start a new journey with VijayTutors !!!

What is Javascript ?

JavaScript is a scripting language used to create and control dynamic website content, i.e. anything that moves, refreshes, or otherwise changes on your screen without requiring you to manually reload a web page. Features like: animated graphics. photo slideshows.

Setting up the environment

Using a browser console and an IDE (like VSCode)

```
console.log("Hello, World!");
```

Variables and Data Types

Varialbles: var, let, const

Datatypes: String, Number, Boolean, Object, Array, Null, Undifined

Example:

```
let name = "Alice"; // String
const age = 30; // Number
var isStudent = true; // Boolean
let scores = [90, 80, 70]; // Array
let person = { firstName: "Alice", lastName: "Smith" }; // Object
let notDefined; // Undefined
let emptyValue = null; // Null
```

Chapter 2

Control Structures

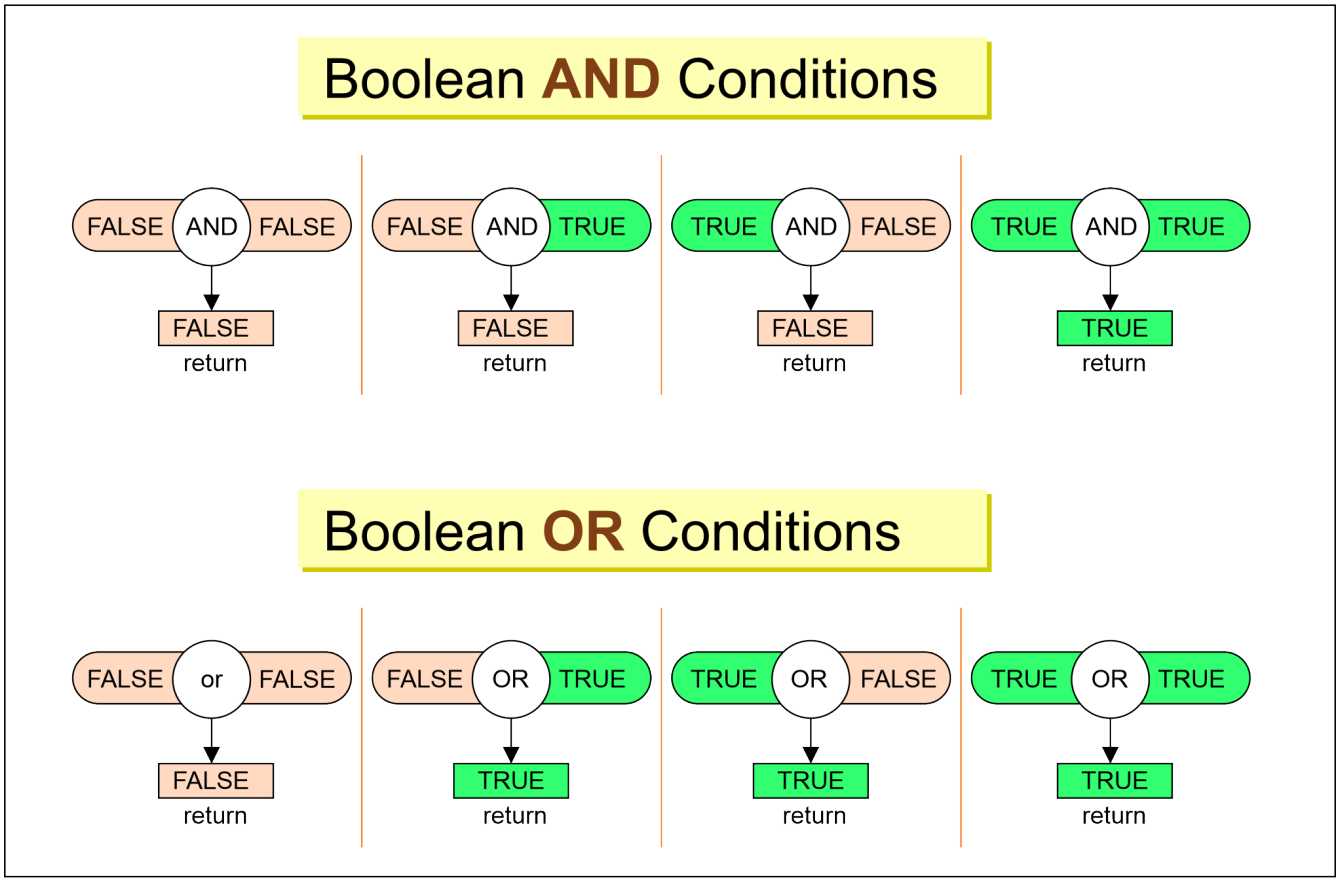
Understanding Conditional Structures in JavaScript

Conditional structures allow your program to make decisions and execute different code based on different conditions. This is crucial in programming for creating dynamic and responsive applications.

Boolean Conditions

In programming and logic, a Boolean is a data type that has two possible values: true and false. The name comes from the mathematician and logician George Boole, who developed Boolean algebra, a branch of algebra that deals with truth values.

Example:



Operators:

JavaScript operators are special symbols that perform operations on operands (values or variables). They are crucial for manipulating data and performing various computations and comparisons in your code. Here's a comprehensive overview of the different types of JavaScript operators:

1. Arithmetic Operators

These operators are used to perform basic mathematical operations:

Addition (+): Adds two values.

```
let sum = 10 + 5; // 15
```

Subtraction (-): Subtracts the second value from the first.

```
let difference = 10 - 5; // 5
```

Multiplication (*): Multiplies two values.

```
let product = 10 * 5; // 50
```

Division (/): Divides the first value by the second.

```
let quotient = 10 / 5; // 2
```

Modulus (%): Returns the remainder of a division operation.

```
let remainder = 10 % 3; // 1
```

Exponentiation (**): Raises the first value to the power of the second.

```
let power = 2 ** 3; // 8
```

Increment (++): Increases a value by one.

```
let count = 1;
count++; // 2
```

Decrement (--): Decreases a value by one.

```
let count = 2;
count--; // 1
```

2. Assignment Operators

Assignment operators is used to assign a value to a variable or property. The assignment expression itself has a value, which is the assigned value. This allows multiple assignments to be chained in order to assign a single value to multiple variables.

Assignment (=): Assigns a value to a variable.

```
let x = 10;
```

Addition Assignment (+=): Adds a value to a variable and assigns the result.

```
let x = 10;
x += 5; // x is now 15
```

Subtraction Assignment (-=): Subtracts a value from a variable and assigns the result.

```
let x = 10;
x -= 5; // x is now 5
```

Multiplication Assignment (*=): Multiplies a variable by a value and assigns the result.

```
let x = 10;
x *= 5; // x is now 50
```

Division Assignment (/=): Divides a variable by a value and assigns the result.

```
let x = 10;
x /= 5; // x is now 2
```

Modulus Assignment (%=): Takes the modulus of a variable and a value and assigns the result.

```
let x = 10;
x %= 3; // x is now 1
```

3. Comparison Operators

A comparison operator compares its operands and returns a logical value based on whether the comparison is true. The operands can be numerical, string, logical, or object values. Strings are compared based on standard lexicographical ordering, using Unicode values.

These operators compare values and return a Boolean result (true or false):

Equality (==): Checks if two values are equal (type coercion occurs).

```
let isEqual = (5 == '5'); // true
```

Strict Equality (===): Checks if two values are equal and of the same type (no type coercion).

```
let isEqual = (5 === '5'); // false
```

Inequality (!=): Checks if two values are not equal (type coercion occurs).

```
let isNotEqual = (5 != '5'); // false
```

Strict Inequality (!==): Checks if two values are not equal or not of the same type (no type coercion).

```
let isNotEqual = (5 !== '5'); // true
```

Greater Than (>): Checks if a value is greater than another.

```
let isGreater = (10 > 5); // true
```

Greater Than or Equal To (>=): Checks if a value is greater than or equal to another.

```
let isGreaterOrEqual = (10 >= 10); // true
```

Less Than (<): Checks if a value is less than another.

```
let isLess = (5 < 10); // true
```

Less Than or Equal To (<=): Checks if a value is less than or equal to another.

```
let isLessOrEqual = (5 <= 5); // true
```

4. Logical Operators

These operators combine Boolean values:

AND (&&): Returns true if both operands are true.

```
let result = (true && false); // false
```

OR (||): Returns true if at least one of the operands is true.

```
let result = (true || false); // true
```

NOT (!): Returns the opposite Boolean value of the operand.

```
let result = !true; // false
```

5. Bitwise Operators

These operate on the binary representations of numbers:

AND (&): Performs a bitwise AND.

```
let result = 5 & 3; // 1
```

OR (|): Performs a bitwise OR.

```
let result = 5 | 3; // 7
```

XOR (^): Performs a bitwise XOR.

```
let result = 5 ^ 3; // 6
```

Complement (~): Inverts the bits.

```
let result = ~5; // -6
```

Left Shift (<<): Shifts bits to the left.

```
let result = 5 << 1; // 10
```

Right Shift (>>): Shifts bits to the right.

```
let result = 5 >> 1; // 2
```

Unsigned Right Shift (>>>): Shifts bits to the right without sign extension.

```
let result = -5 >>> 1; // 2147483642
```

6. Conditional (Ternary) Operator

A shorthand for if-else statements:

Ternary (? :): Evaluates a condition and returns one of two values.

```
let age = 18;
let message = (age >= 18) ? 'Adult' : 'Minor'; // 'Adult'
```

7. Type Operators

These operators help in determining or converting types:

typeof: Returns the type of a variable.

```
let type = typeof 42; // 'number'
```

instanceof: Tests whether an object is an instance of a constructor function.

```
let isArray = [1, 2, 3] instanceof Array; // true
```

8. Spread and Rest Operators

Used in function parameters and array literals:

Spread (...): Expands an iterable (e.g., array) into individual elements.

```
let numbers = [1, 2, 3];
let moreNumbers = [...numbers, 4, 5]; // [1, 2, 3, 4, 5]
```

Rest (...): Gathers multiple elements into an array.

```
function sum(...numbers) {
  return numbers.reduce((acc, num) => acc + num, 0);
}
let total = sum(1, 2, 3); // 6
```

These operators form the backbone of most operations in JavaScript, enabling you to handle a wide variety of tasks and implement complex logic in your code.

Chapter 3

Conditional Structures

Conditional structures in JavaScript are constructs that allow you to execute different blocks of code based on certain conditions. They help control the flow of a program and make decisions. Here's a breakdown of the primary conditional structures in JavaScript:

1. if Statement

The if statement executes a block of code if a specified condition evaluates to true.

```
if (condition) {
    // Code to be executed if condition is true
}
```

Example:

```
let age = 18;
if (age >= 18) {
    console.log("You are an adult.");
}
```

2. If-Else Statements

The if...else statement executes one block of code if the condition is true, and another block of code if the condition is false.

```
if (condition) {
  // code to execute if condition is true
} else if (anotherCondition) {
  // code to execute if anotherCondition is true
} else {
  // code to execute if all conditions are false
}
```

Example: Online Store Discount

Imagine an online store that offers different discounts based on the user's membership status.

```
let membership = "gold"; // can be "silver", "gold", or "none"
let discount;
if (membership === "gold") {
  discount = 20;
} else if (membership === "silver") {
  discount = 10;
} else {
  discount = 0;
}
console.log(`You get a ${discount}% discount.`);
// You get a 20% discount.
```

Explanation:

- If the user has a "gold" membership, they get a 20% discount.
- If the user has a "silver" membership, they get a 10% discount.
- If the user has no membership, they get no discount.

Use Case:

This can be used in e-commerce websites to apply discounts based on user membership levels.

3. switch Statement

The switch statement allows you to select one of many code blocks to be executed based on the value of a variable or expression. It's often used when you have multiple possible values for a variable.

Syntax:

```
switch (expression) {
    case value1:
        // Code to be executed if expression matches value1
```

```
        break;
    case value2:
        // Code to be executed if expression matches value2
        break;
    // More cases...
    default:
        // Code to be executed if expression does not match any case
}
```

Example: Traffic Light System

```
let trafficLight = "red";
switch (trafficLight) {
  case "red":
    console.log("Stop!");
    break;
  case "yellow":
    console.log("Get ready to stop.");
    break;
  case "green":
    console.log("Go!");
    break;
  default:
    console.log("Invalid traffic light color.");
}
```

Explanation:

- If the traffic light is "red", the output is "Stop!".
- If the traffic light is "yellow", the output is "Get ready to stop."
- If the traffic light is "green", the output is "Go!".
- If the traffic light is any other color, the output is "Invalid traffic light color."

Use Case:

This can be used in simulations or games to control traffic light behavior.

4. Ternary Operator

The conditional (ternary) operator is the only JavaScript operator that takes three operands: a condition followed by a question mark (?), then an expression to execute if the condition is truthy followed by a colon (:), and finally the expression to execute if the condition is falsy.

Synatax:

```
condition ? expressionIfTrue : expressionIfFalse;
```

Example: Age Verification

A simple age verification to check if someone is an adult.

```
let age = 20;
let isAdult = age >= 18 ? “Yes” : “No”;
console.log(`Is the person an adult? ${isAdult}`);
```

Explanation:

- If the age is 18 or more, isAdult is set to “Yes”.
- If the age is less than 18, isAdult is set to “No”.

Real World Use Cases

1. Form Validation

Conditional structures are used to validate form inputs. For example, checking if all required fields are filled before submission.

```
let username = “”;
let password = “password123”;

if (username === “” || password === “”) {
  console.log(“All fields must be filled out.”);
} else {
  console.log(“Form submitted successfully.”);
}
```

2. User Authentication

Conditional structures can be used to authenticate users. For example, checking if the entered username and password match the stored credentials.

```
let enteredUsername = “user1”;
let enteredPassword = “password123”;
let storedUsername = “user1”;
let storedPassword = “password123”;

if (enteredUsername === storedUsername && enteredPassword === storedPassword) {
  console.log(“Login successful.”);
} else {
  console.log(“Invalid username or password.”);
}
```

3. Game Logic

Conditional structures are heavily used in game development. For example, checking if a player has won or lost based on their score.

```
let playerScore = 95;
let winningScore = 100;

if (playerScore >= winningScore) {
  console.log(“You win!”);
} else {
  console.log(“Keep trying.”);
}
```

In summary, conditional structures are essential for making decisions in your code, allowing you to execute different blocks of code based on various conditions. Practicing these with real-world examples will help you understand and apply them effectively in your projects.

Chapter 4

JavaScript Functions

JavaScript Functions

Functions in JavaScript are blocks of code designed to perform a particular task. They are executed when they are called (invoked). Here's a deep dive into various types of functions in JavaScript with real-world examples.

1. Function Declaration

A function declaration defines a named function.

syntax:

```
function functionName(parameters) {
  // code to be executed
}
```

Example: Greeting User

```
function greetUser(name) {
  return `Hello, ${name}!`;
}
console.log(greetUser("Alice")); // Output: Hello, Alice!
```

Use Case:

This can be used to create reusable pieces of code that can greet different users.

2. Function Expression

A function expression defines a function as part of a larger expression, typically an assignment.

syntax:

```
const functionName = function(parameters) {
  // code to be executed
};
```

Example: Calculating Area of a Rectangle

```
const calculateArea = function(length, width) {
  return length * width;
};
console.log(calculateArea(5, 3)); // Output: 15
```

Use Case:

Useful when you need to define a function dynamically or pass it as an argument.

3. Arrow Functions

Arrow functions provide a shorthand syntax and do not have their own this.

Syntax:

```
const functionName = (parameters) => {
  // code to be executed
};
```

Example: Doubling Numbers in an Array

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map(number => number * 2);
console.log(doubled); // Output: [2, 4, 6, 8, 10]
```

Use Case:

Ideal for simple, short functions and for maintaining lexical this.

4. Anonymous Functions

Anonymous functions are functions without a name, often used as arguments to other functions.

Syntax:

```
function(parameter) {
  // code to be executed
}
```

Example: Sorting an Array

```
let numbers = [3, 1, 4, 1, 5, 9];
numbers.sort(function(a, b) {
  return a - b;
});
console.log(numbers); // Output: [1, 1, 3, 4, 5, 9]
```

Use Case:

Useful for creating functions on the fly for tasks like sorting or filtering.

5. Immediately Invoked Function Expressions (IIFE)

IIFE are functions that are executed immediately after being defined.

```
(function() {
  // code to be executed immediately
})();
```

Example: Module Pattern

```
(function() {
  let privateVariable = “This is private”;

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

  privateFunction(); // Output: This is private
})();
```

Use Case:

Commonly used for encapsulating code to avoid polluting the global scope.

6. Higher-Order Functions

Functions that take other functions as arguments or return functions as their result.

```
function higherOrderFunction(callback) {
  // code that uses the callback
}
```

Example: Filtering an Array

```
const isEven = number => number % 2 === 0;

const filterArray = (array, callback) => {
  let result = [];
  for (let item of array) {
    if (callback(item)) {
      result.push(item);
    }
  }
  return result;
};

console.log(filterArray([1, 2, 3, 4, 5], isEven)); // Output: [2, 4]
```

Use Case:

Used for operations like filtering, mapping, and reducing arrays.

7. Recursive Functions

Functions that call themselves to solve a problem.

Syntax:

```
function recursiveFunction(parameters) {
  if (condition) {
    // base case
    return value;
  } else {
    // recursive case
    return recursiveFunction(modifiedParameters);
  }
}
```

Example: Factorial Calculation

```
function factorial(n) {
  if (n === 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
console.log(factorial(5)); // Output: 120
```

Use Case:

Used in problems that can be broken down into smaller, similar problems, like calculating factorials or traversing trees.

8. Generator Functions

Functions that can be exited and later re-entered, maintaining their context.

Syntax:

```
function* generatorFunction() {
  // code to be executed
}
```

Example: Fibonacci Sequence

```
function* fibonacci() {
  let [prev, curr] = [0, 1];
  while (true) {
    yield curr;
    [prev, curr] = [curr, prev + curr];
  }
}
const fib = fibonacci();
console.log(fib.next().value); // Output: 1
console.log(fib.next().value); // Output: 1
console.log(fib.next().value); // Output: 2
console.log(fib.next().value); // Output: 3
console.log(fib.next().value); // Output: 5
```

Generator Function Code Explain:

1. Function Declaration with `function*`:

- The * after function denotes that this is a generator function.
- Generator functions are different from regular functions because they can pause and resume execution.

2. Variable Initialization:

```
let [prev, curr] = [0, 1];
```

- This uses array destructuring to initialize prev to 0 and curr to 1, which are the first two numbers in the Fibonacci sequence.

3. Infinite Loop:

```
while (true) { ... }
```

- This loop will run indefinitely, continuously generating the next number in the Fibonacci sequence.

4. Yielding Values:

```
yield curr;
```

- The yield keyword pauses the execution of the generator function and returns the value of **curr** to the caller.
- When the generator is resumed, execution continues from this point.

5. Updating Variables:

```
[prev, curr] = [curr, prev + curr];
```

- This again uses array destructuring to update prev and curr to the next two numbers in the sequence. prev takes the current value of curr, and curr takes the sum of the old values of prev and `curr`.
- *** We will learn more about array and destructuring letter sessions.

Chapter 5

JavaScript Variables

Variables in JavaScript

In JavaScript, variables are used to store data that can be accessed and manipulated throughout your code. You can think of variables as containers for storing data values. Variables are essential for writing dynamic and interactive code.

Types of Variable Declarations

JavaScript provides three ways to declare variables: var, let, and const. Each has its own characteristics and use cases.

1. var

- **Scope:** Variables declared with var are function-scoped. If declared outside a function, they are globally scoped.
- **Hoisting:** var variables are hoisted to the top of their scope, meaning they can be used before they are declared, but the value will be undefined until the declaration is encountered.
- **Re-declaration:** You can re-declare a var variable within the same scope without any errors.

Example:

```
function exampleVar() {
      console.log(a); // undefined (due to hoisting)
      var a = 10;
console.log(a); // 10
}
exampleVar();
```

- **Real-World Use Case:** Before ES6 (ECMAScript 2015), var was the only way to declare variables. It is still used in legacy code, but its use is generally discouraged in favor of let and const due to potential issues with scope and hoisting.

2. let

- **Scope:** Variables declared with let are block-scoped, meaning they are only accessible within the block they are declared in (e.g., inside {}).
- **Hoisting:** let variables are also hoisted, but they are not initialized, so accessing them before declaration results in a ReferenceError.
- **Re-declaration:** You cannot re-declare a let variable in the same scope.

Example:

```
function exampleLet() {
    console.log(b); // ReferenceError: Cannot access 'b' before initialization
    let b = 20;
    console.log(b); // 20
}
exampleLet();
```

- **Real-World Use Case:** let is commonly used when the value of the variable needs to change (i.e., it is mutable) and when you want to limit the variable's scope to a specific block, like in loops or conditionals.

3. const

- Scope: Like let, const variables are block-scoped.
- Hoisting: const variables are hoisted but are not initialized, leading to a ReferenceError if accessed before declaration.
- Re-declaration: You cannot re-declare a const variable in the same scope.
- Immutability: The value of a const variable cannot be reassigned once it is set. However, if the variable holds an object or array, the properties or elements of that object/array can still be modified.

```
function exampleConst() {
    const c = 30;
    console.log(c); // 30
    // c = 40; // TypeError: Assignment to constant variable.
```

```
}
exampleConst();
```

Real-World Use Case: const is used for values that should not change after being assigned, such as configuration values, references to objects, or any data that should remain constant throughout the program.

```
const API_KEY = "12345-ABCDE";
// API_KEY = "67890-FGHIJ"; // TypeError: Assignment to constant variable.

const user = { name: "John" };
user.name = "Jane"; // This is allowed because we are not reassigning the `user`
variable, just modifying its property.
console.log(user.name); // Jane
```

Real-World Example

Consider a scenario where you are developing a web application with user interactions.

```
const userName = "Alice"; // The user's name should not change
let score = 0; // The score will change as the user plays the game
const settings = { theme: "dark", volume: 80 };
// Settings are constant, but their properties can change
function startGame() {
    if (userName) {
        score = 10; // Update score
        settings.volume = 50; // Adjust volume
         console.log(`Welcome ${userName}, your score is ${score} and the volume is
set to ${settings.volume}.`);
    }
}
startGame();
```

In this example:

- userName is a constant because it should not change.
- score is declared with let because it will change.
- settings is a constant reference, but its properties can still be modified.

Summary of Differences:

Features	var	let	const
Scope	function/Global	Block	Block
Hoisting	Yes(initialized as undefined)	Yes (but not initialized)	Yes (but not initialized)
Re-declaration	Allowed	Not allowed	Not allowed
Re-assignment	Allowed	Allowed	Not allowed

Understanding these differences allows you to choose the appropriate variable declaration for your specific needs, leading to more predictable and maintainable code.W

Chapter 6

JavaScript Arrays

What is an Array ?

An array is a special variable, which can hold more than one value

Example:

```
Const colorsUsedInMyWebsite = [“Orange”, “Green”, “White”, “Yellow”, “Black”]
```

JavaScript arrays are versatile and widely used data structures that allow you to store and manipulate collections of data. Here’s an overview of common use cases and frequently used methods in real-world scenarios:

Use Cases of An Array

1. Storing Collections of Data:

- Arrays are often used to store lists of items like names, numbers, objects, etc.
- Example: List of students, products, or tasks.

2. **Iteration:**

- Arrays can be looped through using various iteration methods to perform actions on each item.
- **Example:** Processing a list of orders to calculate the total cost.

3. Data Transformation:

- Arrays can be transformed using methods like map(), filter(), and reduce().
- **Example:** Converting an array of user objects to an array of usernames.

4. Managing Queue or Stack:

- Arrays can function as queues (FIFO) or stacks (LIFO) using methods like `push()`, `pop()`, `shift()`, and `unshift()`.
- Example: Implementing undo functionality or managing a task queue.

5. Data Searching and Sorting:

- Arrays provide methods to search for items and sort them.
- **Example:** Finding a specific product in a catalog or sorting a list of scores.

Frequently Used Array Methods

1. Adding/Removing Elements:

- **push(element1, ..., elementN):** Adds one or more elements to the end of an array.

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

- **pop():** Removes the last element from an array and returns that element.

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

- **shift():** Removes the first element from an array and returns that element.

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

- **unshift(element1, ..., elementN):** Adds one or more elements to the beginning of an array.

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

2. Transforming Arrays:

- **map(callback):** Creates a new array with the results of calling a provided function on every element in the calling array.

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

- **filter(callback):** Creates a new array with all elements that pass the test implemented by the provided function.

```
let arr = [1, 2, 3, 4];
let evens = arr.filter(x => x % 2 === 0);
console.log(evens); // [2, 4]
```

- reduce(callback, initialValue): Executes a reducer function on each element of the array, resulting in a single output value.

```
let arr = [1, 2, 3, 4];
let sum = arr.reduce((acc, x) => acc + x, 0);
console.log(sum); // 10
```

3. Searching:

- **find(callback):** Returns the value of the first element in the array that satisfies the provided testing function.

```
let arr = [1, 2, 3, 4];
let found = arr.find(x => x > 2);
console.log(found); // 3
```

- **findIndex(callback):** Returns the index of the first element in the array that satisfies the provided testing function.

```
let arr = [1, 2, 3, 4];
let index = arr.findIndex(x => x > 2);
console.log(index); // 2
```

- **includes(value):** Determines whether an array includes a certain value among its entries.

4. Manipulating and Inspecting:

- **concat(array1, ..., arrayN):** Merges two or more arrays.

```
let arr1 = [1, 2];
let arr2 = [3, 4];
let merged = arr1.concat(arr2);
console.log(merged); // [1, 2, 3, 4]
```

- **slice(start, end):** Returns a shallow copy of a portion of an array into a new array object.

```
let arr = [1, 2, 3, 4];
let sliced = arr.slice(1, 3);
console.log(sliced); // [2, 3]
```

- **splice(start, deleteCount, item1, ..., itemN):** Changes the contents of an array by removing or replacing existing elements and/or adding new elements.

```
let arr = [1, 2, 3, 4];
arr.splice(1, 2, 'a', 'b');
console.log(arr); // [1, 'a', 'b', 4]
```

- **sort(compareFunction):** Sorts the elements of an array in place and returns the sorted array.

```
let arr = [3, 1, 4, 1, 5, 9];
arr.sort((a, b) => a - b);
console.log(arr); // [1, 1, 3, 4, 5, 9]
```

- **reverse():** Reverses the array in place.

```
let arr = [1, 2, 3];
arr.reverse();
console.log(arr); // [3, 2, 1]
```

5. Iteration:

- **forEach(callback):** Executes a provided function once for each array element.

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

- **every(callback):** Tests whether all elements in the array pass the test implemented by the provided function.

```
let arr = [1, 2, 3];
let allPositive = arr.every(x => x > 0);
console.log(allPositive); // true
```

- **some(callback):** Tests whether at least one element in the array passes the test implemented by the provided function.

```
let arr = [1, 2, 3];
let hasNegative = arr.some(x => x < 0);
console.log(hasNegative); // false
```

Examples of Real-World Use Cases

1. Managing a To-Do List:

```
let tasks = [];
tasks.push({ task: 'Learn JavaScript', completed: false });
tasks.push({ task: 'Write Code', completed: false });
// Mark the first task as completed
tasks[0].completed = true;
// Filter out completed tasks
let incompleteTasks = tasks.filter(task => !task.completed);
console.log(incompleteTasks);
```

2. Processing API Data:

```
let responseData = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' },
  { id: 3, name: 'Charlie' }
];
// Extract names
let names = responseData.map(user => user.name);
console.log(names); // ['Alice', 'Bob', 'Charlie']
```

3. Sorting and Displaying Scores:

```
let scores = [10, 50, 20, 60, 30];
// Sort scores in descending order
scores.sort((a, b) => b - a);
console.log(scores); // [60, 50, 30, 20, 10]
```

By mastering these array methods and understanding their use cases, you'll be well-equipped to handle a wide range of programming tasks in JavaScript.

Array destructuring

Basic Syntax

The basic syntax of array destructuring involves placing the array elements on the left-hand side of the assignment operator =:

```
const [a, b, c] = [1, 2, 3];
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

In this example, the values 1, 2, and 3 from the array [1, 2, 3] are assigned to the variables a, b, and c, respectively.

Skipping Elements

You can skip elements in the array by using commas:

```
const [a, , c] = [1, 2, 3];
console.log(a); // 1
console.log(c); // 3
```

Here, the second element (2) is skipped.

Default Values

You can provide default values for variables in case the array does not contain enough elements:

```
const [a, b = 2, c = 3] = [1];
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

In this case, since the array [1] only has one element, b and c take the default values 2 and 3.

Swapping Variables

Array destructuring can be used to swap the values of variables without a temporary variable:

```
let a = 1;
let b = 2;
[a, b] = [b, a];
console.log(a); // 2
console.log(b); // 1
```

Rest Pattern

The rest pattern (...) can be used to assign the remaining elements of an array to a single variable:

```
const [a, b, ...rest] = [1, 2, 3, 4, 5];
console.log(a); // 1
console.log(b); // 2
console.log(rest); // [3, 4, 5]
```

Nested Destructuring

Destructuring can be nested to handle arrays within arrays:

```
const [a, [b, c]] = [1, [2, 3]];
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

Usage in Functions

Array destructuring is often used in function parameters to handle multiple return values or to work with arrays directly:

```
function example([a, b]) {
  console.log(a);
  console.log(b);
}
example([1, 2]); // 1 2
```

Overall, array destructuring in JavaScript simplifies the process of extracting values from arrays and improves code readability.

Chapter 7

JavaScript Objects

JavaScript Objects: Overview and Examples

JavaScript objects are collections of key-value pairs, where each key (also called a property) is associated with a value. Objects are fundamental to JavaScript and are used to store and manage data.

Basic Structure

Here's the basic syntax for creating an object:

```
const person = {
    firstName: "Vijay",
    lastName: "Mulsaniya",
    age: 40,
    isEmployed: true,
    hobbies: ["reading", "traveling"],
    greet: function() {
        console.log(`Hello, my name is ${this.firstName} ${this.lastName}.`);
    }
};
```

In this example, person is an object with properties firstName, lastName, age, isEmployed, and hobbies, as well as a method greet.

Accessing Object Properties

You can access object properties using dot notation or bracket notation:

```
console.log(person.firstName); // Dot notation
console.log(person['lastName']); // Bracket notation
```

Modifying Object Properties

You can add, update, or delete properties:

```
person.age = 31; // Update
person.email = "john.doe@example.com"; // Add
delete person.isEmployed; // Delete
```

Real-World Use Cases

1. Representing Real-World Entities

Objects are often used to represent real-world entities, such as users, products, or orders.

```
const product = {
    id: 1,
    name: “Laptop”,
    price: 39999.99,
    inStock: true,
    specs: {
        processor: “Intel i7”,
        ram: “16GB”,
        storage: “512GB SSD”
    }
};
```

2. Grouping Related Data

Objects can group related data together, making the code more organized and readable.

```
const userProfile = {
    username: “jdoe”,
    email: “jdoe@example.com”,
    preferences: {
        theme: “dark”,
        language: “en”
    }
};
```

3. Configurations and Settings

Objects are used to manage configurations and settings in applications.

```
const appConfig = {
    apiEndpoint: “https://api.test.com”,
    retryAttempts: 3,
    debugMode: false
};
```

Object Methods

Creating Methods

Objects can have methods (functions that are properties of the object).

```
const calculator = {
    add: function(a, b) {
        return a + b;
    },
    subtract: function(a, b) {
        return a - b;
    }
};

console.log(calculator.add(5, 3)); // 8
console.log(calculator.subtract(5, 3)); // 2
```

this Keyword

The this keyword refers to the current object within a method.

```
const car = {
    brand: “Toyota”,
    model: “Corolla”,
    start: function() {
        console.log(`Starting the ${this.brand} ${this.model}.`);
    }
};

car.start(); // Starting the Toyota Corolla.
```

Iterating Over Object Properties

You can iterate over an object's properties using a for...in loop.

```
const person = {
    firstName: “Jane”,
    lastName: “Smith”,
    age: 28
};

for (let key in person) {
    console.log(`${key}: ${person[key]}`);
}
```

Conclusion

JavaScript objects are versatile and powerful, providing a way to organize and manage data in your applications. Understanding how to use and manipulate objects is essential for effective JavaScript programming.

Chapter 8

JavaScript Events

JavaScript events are actions or occurrences that happen in the browser, and JavaScript can respond to these events by executing specific functions or pieces of code. Events can be triggered by user actions, such as clicking a button, typing in a text field, or moving the mouse, as well as by other interactions like loading a webpage or submitting a form.

Key Concepts of JavaScript Events

Event Types: Here's a comprehensive list of JavaScript events along with brief descriptions:

1. Mouse Events

- **click:** Triggered when an element is clicked.
- **dblclick:** Triggered when an element is double-clicked.
- **mousedown:** Triggered when a mouse button is pressed down on an element
- **mouseup:** Triggered when a mouse button is released over an element.
- **mouseover:** Triggered when the mouse pointer moves over an element.
- **mouseout:** Triggered when the mouse pointer moves out of an element.
- **mousemove:** Triggered when the mouse pointer is moved while over an element.
- **mouseenter:** Triggered when the mouse pointer enters an element.
- **mouseleave:** Triggered when the mouse pointer leaves an element.
- **contextmenu:** Triggered when the right mouse button is clicked (shows a context menu).

2. Keyboard Events

- **keydown:** Triggered when a key is pressed down.
- **keypress:** Triggered when a key is pressed down and released (deprecated; use keydown).
- **keyup:** Triggered when a key is released.

3. Form Events

- **submit:** Triggered when a form is submitted.
- **reset:** Triggered when a form is reset.
- **focus:** Triggered when an element (like an input field) gains focus.
- **blur:** Triggered when an element loses focus.
- **change:** Triggered when the value of an input element is changed and the element loses focus.
- **input:** Triggered every time the value of an input element is changed, regardless of focus.
- **select:** Triggered when some text is selected in a text field.

4. Window Events

- **load:** Triggered when the whole page (including all dependent resources like images) has loaded.
- **unload:** Triggered when the page is unloaded or navigated away from.
- **resize:** Triggered when the window is resized.
- **scroll:** Triggered when the user scrolls within an element or the entire page.
- beforeunload: Triggered just before the page is unloaded, often used to prompt the user if they want to leave the page.
- **error:** Triggered when an error occurs during the loading of an external file.

5. Clipboard Events

- **copy:** Triggered when content is copied to the clipboard.
- **cut:** Triggered when content is cut from the document to the clipboard.
- **paste:** Triggered when content is pasted from the clipboard into the document.

6. Media Events

- play: Triggered when media starts playing.
- pause: Triggered when media is paused.
- ended: Triggered when media has finished playing.
- volumechange: Triggered when the volume of the media changes.
- timeupdate: Triggered when the playing position of the media is updated.

7. Touch Events (for mobile devices)

- **touchstart:** Triggered when a touch point is placed on the touch surface.
- **touchmove:** Triggered when a touch point is moved along the touch surface.
- **touchend:** Triggered when a touch point is removed from the touch surface.
- **touchcancel:** Triggered when a touch point is disrupted, like when the touch surface loses power.
-

8. Drag Events

- **drag:** Triggered while an element is being dragged.
- **dragstart:** Triggered at the start of a drag operation.
- **dragend:** Triggered when a drag operation ends.
- **dragenter:** Triggered when a dragged item enters a valid drop target.
- **dragover:** Triggered when a dragged item is over a valid drop target.
- **dragleave:** Triggered when a dragged item leaves a valid drop target.
- **drop:** Triggered when a dragged item is dropped on a valid drop target.

9. Animation Events

- **animationstart:** Triggered when a CSS animation starts.
- **animationend:** Triggered when a CSS animation ends.
- **animationiteration:** Triggered when a CSS animation completes an iteration.

10. Transition Events

- **transitionstart:** Triggered when a CSS transition starts.
- **transitionend:** Triggered when a CSS transition ends.
- **transitioncancel:** Triggered when a CSS transition is canceled.

11. Focus Events

- **focusin:** Triggered when an element is about to receive focus.
- **focusout:** Triggered when an element is about to lose focus.

12. Pointer Events (combines mouse, touch, and pen events)

- **pointerdown:** Triggered when a pointer is pressed down.
- **pointerup:** Triggered when a pointer is released.
- **pointermove**: Triggered when a pointer is moved.
- **pointerenter:** Triggered when a pointer enters an element.
- **pointerleave:** Triggered when a pointer leaves an element.
- **pointercancel:** Triggered when the pointer device's activity is canceled.

13. Miscellaneous Events

- **contextmenu:** Triggered when the right mouse button is clicked (context menu).
- **hashchange:** Triggered when the URL's hash (#) changes.
- **popstate:** Triggered when the active history entry changes (e.g., when the user navigates back or forward).
- **online:** Triggered when the browser gains network connectivity.
- **offline:** Triggered when the browser loses network connectivity.
- **storage:** Triggered when a storage area (localStorage or sessionStorage) is modified.
- **input:** Triggered when the value of an element is changed, without considering focus.
- **visibilitychange:** Triggered when the visibility state of the page changes (e.g., switching tabs).

These events can be used to create a wide range of interactive web applications, responding to user inputs, browser states, and more.

Event Listeners:

Event Listeners is a functions that are called in response to an event. They are typically added using the addEventListener method.

```
document.getElementById('myButton').addEventListener('click', function() {
    alert('Button clicked!');
});
```

In JavaScript, there are several ways to raise (or trigger) events. Here are the main methods:

1. Using `HTMLElement.click()` Method

- This method triggers a click event programmatically on an element.
- It's commonly used to simulate a user clicking a button.

```
const button = document.getElementById('myButton');
button.click(); // Programmatically triggers a click event
```

2. Using `Event` Constructor

- You can create and dispatch custom events using the Event constructor.
- The Event constructor allows you to create any event by specifying the event type and, optionally, initialization options.

```
const event = new Event('build');
document.dispatchEvent(event); // Raises a 'build' event
```

3. Using `CustomEvent` Constructor

- The CustomEvent constructor is a specialized version of the Event constructor, allowing you to pass additional data (detail) with the event.

```
const event = new CustomEvent('build', { detail: { key: 'value' } });
document.dispatchEvent(event); // Raises a 'build' event with additional data
```

4. Using `dispatchEvent()` Method

- This method is used in conjunction with an event created by either the Event or CustomEvent constructors.
- It triggers the specified event on the element.

```
const event = new Event('build');
document.getElementById('myElement').dispatchEvent(event);
// Raises the 'build' event on a specific element
```

5. Using `setTimeout` or `setInterval`

- While not directly raising an event, setTimeout and setInterval can be used to execute functions that can trigger events after a delay or at intervals.

```
setTimeout(function() {
      const event = new Event('timeoutEvent');
      document.dispatchEvent(event); // Raises the 'timeoutEvent' after a delay
}, 1000);
```

6. Using `EventTarget.dispatchEvent()`

- Similar to dispatchEvent() on DOM elements, this method can be used on any object that implements the EventTarget interface.
- It's used to dispatch an event to the object.

```
const eventTarget = new EventTarget();
const event = new Event('myEvent');
eventTarget.dispatchEvent(event); // Raises 'myEvent' on the EventTarget object
```

7. Simulating Input Events

- For form elements like input, you can set their value programmatically and trigger related events manually.

```
const input = document.getElementById('myInput');
input.value = 'New Value';
input.dispatchEvent(new Event('input')); // Manually triggers an 'input' event
```

Summary

These methods allow you to raise or simulate events in JavaScript, making your application interactive and responsive to both user actions and programmatic triggers. Each method serves different purposes, from simulating user interaction to creating complex, custom events for more advanced scenarios.

Real-World Examples

1. Form Validation:

When a user submits a form, you may want to validate the input fields before sending the data to the server. JavaScript can intercept the submit event to prevent the form from being submitted if certain conditions aren't met.

```
document.getElementById('myForm').addEventListener('submit', function(event) {
    const inputField = document.getElementById('name');
    if (inputField.value === '') {
        alert('Name is required!');
        event.preventDefault(); // Prevents form submission
    }
});
```

2. Interactive Navigation Menu:

When a user hovers over a navigation menu, you can use the mouseover and `mouseout` events to show or hide sub-menus dynamically.

```
document.getElementById('menu').addEventListener('mouseover', function() {
    document.getElementById('submenu').style.display = 'block';
});

document.getElementById('menu').addEventListener('mouseout', function() {
    document.getElementById('submenu').style.display = 'none';
});
```

3. Dynamic Content Loading:

You can load content dynamically when the user scrolls to the bottom of the page using the

scroll event. This is commonly seen in infinite scrolling websites.

```
window.addEventListener('scroll', function() {
    if ((window.innerHeight + window.scrollY) >= document.body.offsetHeight) {
        loadMoreContent(); // Function to load more content
    }
});
```

4. Real-Time Search Filtering:

When a user types into a search field, you can use the keyup event to filter a list of items in real time.

```
document.getElementById('search').addEventListener('keyup', function() {
    const filter = this.value.toLowerCase();
    const items = document.querySelectorAll('.item');
    items.forEach(function(item) {
        if (item.textContent.toLowerCase().includes(filter)) {
            item.style.display = '';
        } else {
            item.style.display = 'none';
        }
    });
});
```

6. Custom Events:

Sometimes, you may want to create your own events to manage complex interactions. For example, in a gaming application, you might want to trigger an event when a player wins the game.

```
const event = new Event('gameWon');
document.addEventListener('gameWon', function() {
    alert('You won the game!');
});

// Later in the code, when the game is won:
document.dispatchEvent(event);
```

Conclusion

JavaScript events are essential for creating interactive, dynamic web applications. By understanding and using events, you can respond to user actions and make your website more responsive and engaging. Whether you're handling form submissions, mouse movements, or custom game events, mastering JavaScript events will significantly enhance your web development skills.

Notes:

Chapter 9

JavaScript String Methods

JavaScript has a rich set of string methods that can be very useful for manipulating and interacting with strings. Let's go over some of the most commonly used string methods with real-world examples:

1. charAt(index)

Returns the character at a specified index.

```
let str = "Hello, World!";
console.log(str.charAt(7)); // Outputs: W
```

Use Case: You might use this method when you need to get a specific character from a string.

2. concat(...strings)

Combines multiple strings into one.

```
let str1 = "Hello";
let str2 = "World";
console.log(str1.concat(", ", str2, "!")); // Outputs: Hello, World!
```

Use Case: Useful for combining several strings into a single string.

3. includes(substring)

Checks if a substring exists within a string.

```
let str = "JavaScript is awesome!";
console.log(str.includes("awesome")); // Outputs: true
```

Use Case: Ideal for checking if a certain word or phrase is present in a string.

4. indexOf(substring)

Returns the index of the first occurrence of a substring.

```
let str = "Find the index of this word.";
console.log(str.indexOf("index")); // Outputs: 5
```

Use Case: Useful for finding the position of a substring within a string.

5. lastIndexOf(substring)

Returns the index of the last occurrence of a substring.

```
let str = "Find the last index of the word index.";
console.log(str.lastIndexOf("index")); // Outputs: 30
```

Use Case: Useful when you need to find the last occurrence of a substring.

6. replace(searchValue, newValue)

Replaces the first occurrence of a substring or pattern with a new string.

```
let str = "Hello, old friend!";
console.log(str.replace("old", "new")); // Outputs: Hello, new friend!
```

Use Case: Handy for substituting a part of a string with a new value.

7. slice(start, end)

Extracts a portion of the string based on start and end indices.

```
let str = "Extract a part of this string.";
console.log(str.slice(10, 23)); // Outputs: part of this
```

Use Case: Useful for getting a substring based on specific positions.

8. split(separator)

Splits a string into an array of substrings based on a separator.

```
let str = "JavaScript,Python,Ruby";
let arr = str.split(",");
console.log(arr); // Outputs: ["JavaScript", "Python", "Ruby"]
```

Use Case: Ideal for breaking a string into parts, especially when working with delimited data.

9. toLowerCase() and toUpperCase()

Converts the entire string to lowercase or uppercase.

```
let str = "Case Insensitive";
console.log(str.toLowerCase()); // Outputs: case insensitive
console.log(str.toUpperCase()); // Outputs: CASE INSENSITIVE
```

Use Case: Useful for normalizing strings for case-insensitive comparisons or formatting.

10. trim()

Removes whitespace from both ends of the string.

```
let str = "   Trim me!   ";
console.log(str.trim()); // Outputs: Trim me!
```

Use Case: Handy for cleaning up user input or data read from a file.

11. repeat(count)

Returns a new string with the specified number of copies of the original string.

```
let str = "Ha!";
console.log(str.repeat(3)); // Outputs: Ha!Ha!Ha!
```

Use Case: Useful for creating repeated patterns or simple animations.

12. startsWith(searchString, position)

Determines whether a string starts with a given substring.

```
let str = "Hello, World!";
console.log(str.startsWith("Hello")); // Outputs: true
```

Use Case: Helpful for checking if a string begins with a certain prefix.

13. endsWith(searchString, length)

Determines whether a string ends with a given substring.

```
let str = "Hello, World!";
console.log(str.endsWith("World!")); // Outputs: true
```

Use Case: Useful for checking if a string ends with a certain suffix.

14. match(regex)

Retrieves the result of matching a string against a regular expression.

```
let str = “The rain in Spain stays mainly in the plain.”;
let result = str.match(/ain/g);
console.log(result); // Outputs: [“ain”, “ain”, “ain”]
```

Use Case: Useful for finding all matches of a pattern in a string.

15. search(regex)

Executes a search for a match between a string and a regular expression.

```
let str = “The rain in Spain.”;
let index = str.search(/Spain/);
console.log(index); // Outputs: 12
```

Use Case: Useful for finding the index of the first match of a pattern.

These methods cover a broad range of common operations you might need to perform on strings. Experimenting with them in your projects will help solidify your understanding.

In next chapter we will learn more about regex

Chapter 10

Regular Expressions (regex)

Regular expressions (often abbreviated as regex or regexp) are a powerful tool used for pattern matching and manipulation within strings. They are a sequence of characters that define a search pattern, usually for matching, searching, or replacing text. Regular expressions are supported in many programming languages, including JavaScript, Python, Java, and more.

Key Concepts of Regular Expressions

1. Literal Characters

- Simple characters that match themselves.
- Example: hello matches the exact string “hello”.

2. Metacharacters

Special characters that have a specific meaning in regex:

- **.** : Matches any single character except newline.

  Example: a.c matches **“abc”, “a2c”, “a-c”,** etc.

- **^** : Matches the start of a string.

  Example: ^hello matches **“hello world”** but not **“world hello”**.

- **$** : Matches the end of a string.

  Example: world$ matches **“hello world”** but not **“world hello”**.

- ***** : Matches zero or more occurrences of the preceding element.

  Example: a* matches **“”, “a”, “aa”, “aaa”**, etc.

- **+** : Matches one or more occurrences of the preceding element.

  Example: a+ matches **“a”, “aa”, “aaa”**, but not **“”**.

- ?: Matches zero or one occurrence of the preceding element.

  Example: a? matches **“”** or **“a”**.

- **{n}** : Matches exactly n occurrences of the preceding element.

  Example: a{3} matches **“aaa”**.

- **{n,}:** Matches n or more occurrences of the preceding element.

  Example: a{2,} matches **“aa”, “aaa”, “aaaa”,** etc.

- {n,m}: Matches between n and m occurrences of the preceding element.

  Example: a{2,4} matches **“aa”, “aaa”**, or **“aaaa”**.

- | : Acts as an OR operator.

  Example: cat|dog matches **“cat”** or **“dog”.**

3. Character Classes

Define a set of characters to match.

- **[abc]:** Matches any one of the characters **a, b, or c**.
- **[^abc]:** Matches any character except **a, b,** or **c**.
- **[a-z]:** Matches any lowercase letter from **a** to **z**.
- **[0-9]:** Matches any digit.

4. Predefined Character Classes

- **\d:** Matches any digit (equivalent to **[0-9]**).
- **\D:** Matches any non-digit character.
- **\w:** Matches any word character (alphanumeric plus underscore, equivalent to **[a-zA-Z0-9_]**).
- **\W:** Matches any non-word character.
- **\s:** Matches any whitespace character (spaces, tabs, line breaks).
- **\S:** Matches any non-whitespace character.

5. Grouping and Capturing

- **() :** Groups part of the regex.

  Example: (abc)+ matches **“abc”, “abcabc”**, etc.

- **(?:...)** : Non-capturing group (groups part of the regex without capturing).
- **(abc|def)**: Matches either **“abc”** or **“def”**.

6. Escaping Special Characters

- ****: Escapes a metacharacter to match it literally.

  Example: \$ matches the dollar sign **“ $”**.

Examples of Regular Expressions in JavaScript

1. Matching a Simple Pattern

```
let regex = /hello/;
console.log(regex.test(“hello world”)); // Outputs: true
```

2. Extracting Data

```
let text = “Email me at example@example.com”;
let match = text.match(/[\w.-]+@[\w.-]+\.\w+/);
console.log(match[0]); // Outputs: example@example.com
```

3. Replacing Text

```
let text = “I love apples and bananas.”;
let newText = text.replace(/apples|bananas/g, “fruit”);
console.log(newText); // Outputs: I love fruit and fruit.
```

4. Splitting a String

```
let csv = “apple,orange,banana,grape”;
let fruits = csv.split(/,\s*/);
console.log(fruits); // Outputs: [“apple”, “orange”, “banana”, “grape”]
```

Regular expressions can be complex and powerful, allowing for intricate text processing tasks. They might seem intimidating at first, but once you get the hang of the syntax and concepts, they become a valuable tool for handling strings.

Basic Email Validation Regex

Here's a commonly used regular expression for validating email addresses:

```
/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/
1       2        3  4          5  6       7
```

Explanation of the Regex

- **^** : Asserts the position at the start of the string.
- **[a-zA-Z0-9._%+-]+**
- **[a-zA-Z0-9._%+-]** : Matches any letter (both uppercase and lowercase), digit, dot (.), underscore (_), percent (%), plus (+), or hyphen (-).
- **+** : Ensures that the preceding character class appears one or more times. This matches the local part of the email (before the @ symbol).
- **@** : Matches the @ symbol literally.
- **[a-zA-Z0-9.-]+**
- **[a-zA-Z0-9.-]** : Matches any letter (both uppercase and lowercase), digit, dot (.), or hyphen (-).
- **+** : Ensures that the preceding character class appears one or more times. This matches the domain part of the email (after the @ symbol).
- **\.** : Matches a literal dot (.) that separates the domain from the top-level domain.
- **[a-zA-Z]{2,}**
- **[a-zA-Z]** : Matches any letter (both uppercase and lowercase).
- **{2,}** : Ensures that the preceding character class appears at least 2 times. This matches the top-level domain (like com, org, net, etc.).
- **$** : Asserts the position at the end of the string.

Example Code in JavaScript

Here's how you might use this regex pattern in JavaScript to validate an email address:

```
function validateEmail(email) {
    const regex = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
    return regex.test(email);
}
// Test cases
console.log(validateEmail("example@example.com")); // true
console.log(validateEmail("user.name+tag+sorting@example.com")); // true
console.log(validateEmail("user@subdomain.example.co.uk")); // true
console.log(validateEmail("invalid-email@.com")); // false
console.log(validateEmail("invalid-email@com")); // false
console.log(validateEmail("invalid-email@com.")); // false
```

Notes

- This regex pattern covers most common email formats but does not cover all possible valid email formats according to the official standards (RFC 5321 and RFC 5322).
- The pattern doesn't handle cases such as quoted strings in the local part of the email or comments.
- For more comprehensive validation, especially to cover edge cases, you might need a more complex regex or use built-in email validation methods provided by frameworks or libraries.

This regex should be sufficient for most practical purposes where you need to validate typical email addresses.

Chapter 11
JavaScript Dates

JavaScript Dates can be quite useful for handling and manipulating dates and times in your web applications. Let's dive into the basics, some common methods, and real-world use cases.

Basics of JavaScript Dates

In JavaScript, you use the Date object to work with dates and times. Here's how you can create and manipulate dates:

1. Creating a Date Object

```
// Current date and time
const now = new Date();
console.log(now);
// Example output: Fri Aug 16 2024 18:53:44 GMT+0530 (India Standard Time)

// Specific date and time
const specificDate = new Date('2024-08-15T12:34:56Z');
console.log(specificDate);
// Example output: Thu Aug 15 2024 18:04:56 GMT+0530 (India Standard Time)

// Date with year, month, day, hour, minute, second, millisecond
const customDate = new Date(2024, 7, 15, 12, 34, 56); // Months are 0-indexed
console.log(customDate);
// Example output: Thu Aug 15 2024 12:34:56 GMT+0530 (India Standard Time)
```

2. Getting Date and Time Information

```
// Get the current year
console.log(now.getFullYear()); // Example output: 2024

// Get the current month (0-indexed)
console.log(now.getMonth()); // Example output: 7 (August)
```

```
// Get the current day of the month
console.log(now.getDate()); // Example output: 16

// Get the current day of the week
console.log(now.getDay()); // Example output: 5 (Friday)

// Get the current hours
console.log(now.getHours()); // Example output: 18

// Get the current minutes
console.log(now.getMinutes()); // Example output: 58

// Get the current seconds
console.log(now.getSeconds()); // Example output: 31
```

3. Setting Date and Time Information

```
const date = new Date();

// Set the year
date.setFullYear(2025);
console.log(date);
// Example output: Sat Aug 16 2025 19:02:39 GMT+0530 (India Standard Time)

// Set the month (0-indexed)
date.setMonth(11); // December
console.log(date);
// Example output: Tue Dec 16 2025 19:02:39 GMT+0530 (India Standard Time)

// Set the day of the month
date.setDate(25);
console.log(date);
// Example output: Thu Dec 25 2025 19:02:39 GMT+0530 (India Standard Time)
```

```
// Set the hours
date.setHours(9);
console.log(date);
// Example output: Thu Dec 25 2025 09:02:39 GMT+0530 (India Standard Time)
```

4. Date Calculations

```
const now = new Date();

// Adding 1 day
const tomorrow = new Date(now);
tomorrow.setDate(now.getDate() + 1);
console.log(tomorrow);

// Subtracting 1 month
const lastMonth = new Date(now);
lastMonth.setMonth(now.getMonth() - 1);
console.log(lastMonth);
```

Real-World Use Cases

1. Displaying Current Time on a Web Page

You might want to show the current time or date on your website.

```
<p id="current-time"></p>
<script>
  function updateTime() {
    const now = new Date();
    document.getElementById('current-time').textContent = now.toLocaleString();
  }
  updateTime();
  setInterval(updateTime, 1000); // Update every second
</script>
```

2. Form Handling

Validating dates in forms, such as checking if a user's input is a valid date or if a date is within a certain range.

```
function validateDate(inputDate) {
  const now = new Date();
  const enteredDate = new Date(inputDate);

  if (enteredDate < now) {
    console.log('Date is in the past');
  } else {
    console.log('Date is valid');
  }
}
validateDate('2024-08-14'); // Example output: Date is in the past
```

3. Scheduling and Reminders

Creating reminders or scheduling tasks based on dates and times.

```
function scheduleReminder(date, message) {
      const now = new Date();
      const reminderDate = new Date(date);
      if (reminderDate > now) {
            const timeout = reminderDate - now;
            setTimeout(() => alert(message), timeout);
      } else {
        WWconsole.log('Reminder date is in the past');
      }
}
scheduleReminder('2024-08-16T19:49:00', 'Reminder: Meeting at 9 AM!');
```

Chapter 12

JavaScript Math Object

JavaScript Math object is a built-in object that provides properties and methods for mathematical constants and functions. Here's a rundown of some frequently used methods and properties of the Math object, along with examples and typical use cases:

1. Math.abs()

Description: Returns the absolute value of a number.

Use Case: Useful when you need to get a non-negative value regardless of the sign.

```
console.log(Math.abs(-5)); // 5
console.log(Math.abs(3.14)); // 3.14
```

2. Math.ceil()

Description: Rounds a number UP to the nearest integer.

Use Case: Useful when you need to ensure a number rounds up, for example in cases like pagination where you need to round up to the next whole page.

```
console.log(Math.ceil(4.2)); // 5
console.log(Math.ceil(-4.7)); // -4
```

3. Math.floor()

Description: Rounds a number DOWN to the nearest integer.

Use Case: Useful for rounding down in scenarios like creating indexes or determining discrete values.

```
console.log(Math.floor(4.7)); // 4
console.log(Math.floor(-4.2)); // -5
```

4. Math.round()

Description: Rounds a number to the nearest integer.

Use Case: Commonly used to round numbers to the nearest whole number, like when dealing with prices or scores.

```
console.log(Math.round(4.5)); // 5
console.log(Math.round(4.4)); // 4
```

5. Math.max()

Description: Returns the largest of zero or more numbers.

Use Case: Useful for determining the maximum value from a set of numbers.

```
console.log(Math.max(1, 3, 2)); // 3
console.log(Math.max(-1, -3, -2)); // -1
```

6. Math.min()

Description: Returns the smallest of zero or more numbers.

Use Case: Useful for finding the minimum value from a set of numbers.

```
console.log(Math.min(1, 3, 2)); // 1
console.log(Math.min(-1, -3, -2)); // -3
```

7. Math.random()

Description: Returns a pseudo-random floating-point number in the range from 0 to 1

Use Case: Commonly used for generating random numbers, such as for games or sampling data. (We will discuss more about this function. see page no: 60)

```
console.log(Math.random()); // e.g., 0.123456789
```

8. Math.pow(base, exponent)

Description: Returns the base to the exponent power.

Use Case: Useful for mathematical calculations involving exponents.

```
console.log(Math.pow(2, 3)); // 8
console.log(Math.pow(5, 2)); // 25
```

9. Math.sqrt()

Description: Returns the square root of a number.

```
console.log(Math.sqrt(16)); // 4
console.log(Math.sqrt(25)); // 5
```

10. Math.sin(), Math.cos(), Math.tan()

Description: Returns the trigonometric sine, cosine, and tangent of a number (in radians).

Use Case: Useful for calculations involving angles, such as in graphics programming or physics simulations.

```
console.log(Math.sin(Math.PI / 2)); // 1
console.log(Math.cos(0)); // 1
console.log(Math.tan(Math.PI / 4)); // 1
```

11. Math.log()

Description: Returns the natural logarithm (base e) of a number.

```
console.log(Math.log(1)); // 0
console.log(Math.log(Math.E)); // 1
```

12. Math.trunc()

Description: Returns the integer part of a number by removing any fractional digits.

Use Case: Useful for extracting the whole number portion of a floating-point number.

```
console.log(Math.trunc(4.9)); // 4
console.log(Math.trunc(-4.9)); // -4
```

13. Math.sign()

Description: Returns the sign of a number, indicating whether it is positive, negative, or zero.

Use Case: Useful for determining the direction or sign of a number.

```
console.log(Math.sign(5)); // 1
console.log(Math.sign(-5)); // -1
```

14. Math.PI

Description: Provides the value of π (pi), approximately 3.14159.

Use Case: Useful for calculations involving circles and other geometrical figures.

```
console.log(Math.PI); // 3.141592653589793
```

These methods and properties make it easier to perform a variety of mathematical operations and handle numerical data effectively in JavaScript.

Basic Usage of Math.random()

The Math.random() function in JavaScript is used to generate a pseudo-random floating-point number in the range from 0 (inclusive) to 1 (exclusive). This means the result will be a decimal number greater than or equal to 0 but less than 1.

```
let randomNumber = Math.random();
console.log(randomNumber); // e.g., 0.5378442292611035
```

Real-World Examples

1. Randomizing Elements in an Array

Suppose you want to shuffle elements in an array to ensure they appear in a random order. Here's how you can use Math.random() for that:

```
function shuffleArray(array) {
  for (let i = array.length - 1; i > 0; i--) {
    const j = Math.floor(Math.random() * (i + 1));
    [array[i], array[j]] = [array[j], array[i]]; // Swap elements
  }
  return array;
}
const array = [1, 2, 3, 4, 5];
console.log(shuffleArray(array)); // e.g., [3, 1, 5, 2, 4]
```

2. Simulating Dice Rolls

To simulate rolling a six-sided die, you can use Math.random() to generate a number between 1 and 6:

```
function rollDice() {
  return Math.floor(Math.random() * 6) + 1; // Generates a number between 1 and 6
}
console.log(rollDice()); // e.g., 4
```

3. Generating Random Passwords

For generating random passwords or codes, you can combine Math.random() with other string operations:

```
function generateRandomPassword(length) {
 const characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
  let password = '';
  for (let i = 0; i < length; i++) {
    const randomIndex = Math.floor(Math.random() * characters.length);
    password += characters[randomIndex];
  }
  return password;
}
console.log(generateRandomPassword(8)); // e.g., 'a3Jk9Dfs'
```

4. Generating Random Numbers in a Specified Range (between 1 and 100)

To get a random integer, you can use Math.floor() to round down to the nearest whole number:

```
function getRandomIntInRange(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
console.log(getRandomIntInRange(1, 100)); // e.g., 42
```

Summary

- Math.random() generates a floating-point number between 0 and 1.
- To get a number in a specific range, scale and shift the result of Math.random().
- Use Math.floor() to round down for integers if needed.

These methods are quite useful for a variety of applications, including games, simulations, and data generation.

Chapter 13

Error handling

Error handling is crucial in JavaScript to ensure that your code runs smoothly and to manage unexpected situations gracefully. JavaScript provides several ways to handle errors, primarily using try...catch, throw, and finally blocks. Let's explore these concepts with real-world examples.

1. try...catch Statement

The try...catch statement is used to execute a block of code and catch any errors that occur during execution. It allows you to handle errors and prevent them from crashing your application.

Syntax:

```
try {
  // Code that may throw an error
} catch (error) {
  // Code that runs if an error occurs
}
```

Example 1: Handling Network Request Errors

Imagine you're making a network request to fetch user data. If the request fails, you want to handle the error gracefully and provide feedback to the user.

```
function fetchUserData(userId) {
  try {
    // Simulate a network request
    if (userId <= 0) {
      throw new Error('Invalid user ID');
    }
    console.log('Fetching data for user ID:', userId);
    // Simulate a successful response
    return { id: userId, name: 'John Doe' };
  } catch (error) {
    console.error('Error fetching user data:', error.message);
    // Handle the error (e.g., show an alert or default data)
    return null;
  }
}
```

```
const userData = fetchUserData(-1); // Invalid user ID
console.log(userData); // null
```

2. throw Statement

The throw statement is used to create custom errors. You can throw errors with a custom message to indicate specific problems in your code.

Syntax:

```
throw new Error('Error message');
```

Example 2: Validating User Input

Suppose you have a function that calculates the square root of a number but needs to validate that the number is non-negative. If the input is invalid, you throw an error.

```
function calculateSquareRoot(value) {
  if (value < 0) {
    throw new Error('Cannot calculate the square root of a negative number');
  }
  return Math.sqrt(value);
}

try {
  const result = calculateSquareRoot(-9);
  console.log('Square root:', result);
} catch (error) {
  console.error('Error:', error.message);
}
```

3. finally Block

The finally block is optional and, if used, will execute after the try and catch blocks, regardless of whether an error occurred. It's typically used for cleanup operations.

Syntax:

```
try {
  // Code that may throw an error
} catch (error) {
  // Code that runs if an error occurs
} finally {
  // Code that always runs
}
```

Example 3: Cleaning Up Resources

Imagine you have a function that opens a file, processes it, and needs to ensure the file is closed whether or not an error occurs.

```
function processFile(fileName) {
  let fileHandle;
  try {
    fileHandle = openFile(fileName); // Hypothetical function to open a file
    console.log('Processing file:', fileName);
    // Process the file
  } catch (error) {
    console.error('Error processing file:', error.message);
  } finally {
    if (fileHandle) {
      closeFile(fileHandle); // Hypothetical function to close the file
      console.log('File closed.');
    }
  }
}
processFile('example.txt');
```

4. Error Object Properties

When handling errors, the Error object contains useful properties:

- message: A description of the error.
- name: The type of error (e.g., Error, TypeError, SyntaxError).
- stack: A stack trace that shows where the error occurred (useful for debugging).

Example 4: Custom Error Messages and Types

You can create custom error types by extending the Error class:

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = 'ValidationError';
  }
}
function validateAge(age) {
  if (age < 0) {
    throw new ValidationError('Age cannot be negative');
  }
  return age;
}
try {
  const age = validateAge(-5);
  console.log('Age:', age);
} catch (error) {
  if (error instanceof ValidationError) {
    console.error('Validation Error:', error.message);
  } else {
    console.error('General Error:', error.message);
  }
}
```

Summary

- **try...catch**: Handle errors during code execution.
- **throw**: Create and throw custom errors.
- **finally**: Execute code after try and catch, regardless of errors.
- **Error Object**: Provides information about the error, including message, name, and stack.

Error handling allows you to manage unexpected situations and maintain a smooth user experience in your applications.

Chapter 14

JavaScript Class

What is a JavaScript Class?

A JavaScript class is a blueprint for creating objects. It provides a way to define objects with shared properties and methods. Classes are syntactical sugar over JavaScript's existing prototype-based inheritance and are introduced in ECMAScript 6 (ES6).

Basic Structure of a Class

Here's the basic structure of a class:

```
class MyClass {
  constructor(property1, property2) {
    this.property1 = property1;
    this.property2 = property2;
  }

  method1() {
    console.log('Method 1');
  }
  method2() {
    console.log('Method 2');
  }
}
```

Real-World Examples

1. Creating a Car Class

Imagine you want to create different car objects with shared properties and behaviors.

```
class Car {
  constructor(make, model, year) {
    this.make = make;
    this.model = model;
    this.year = year;
  }
```

```
  startEngine() {
    console.log(${this.make} ${this.model}'s engine started.);
  }
  stopEngine() {
    console.log(${this.make} ${this.model}'s engine stopped.);
  }
  getDetails() {
    return ${this.year} ${this.make} ${this.model};
  }
}
const myCar = new Car('Toyota', 'Corolla', 2021);
myCar.startEngine(); // Toyota Corolla's engine started.
console.log(myCar.getDetails()); // 2021 Toyota Corolla
```

In this example, Car is a class with a constructor that initializes the car's make, model, and year. It also has methods to start the engine, stop the engine, and get the car's details.

2. Creating a User Class

Let's create a User class that can be used to handle user information.

```
class User {
  constructor(username, email) {
    this.username = username;
    this.email = email;
  }

  greet() {
    console.log(Hello, ${this.username}!);
  }

  updateEmail(newEmail) {
    this.email = newEmail;
    console.log(Email updated to ${this.email});
  }
}
```

```
const user1 = new User('john_doe', 'john@example.com');
user1.greet(); // Hello, john_doe!
user1.updateEmail('john.doe@newdomain.com');
// Email updated to john.doe@newdomain.com
```

Here, the User class includes methods to greet the user and update their email address.

3. Inheritance with Classes

You can create subclasses that inherit from a parent class. This allows you to extend and modify functionality.

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(${this.name} makes a noise.);
  }
}

class Dog extends Animal {
  speak() {
    console.log(${this.name} barks.);
  }
}


class Cat extends Animal {
  speak() {
    console.log(${this.name} meows.);
  }
}

const myDog = new Dog('Rex');
const myCat = new Cat('Whiskers');
```

```
myDog.speak(); // Rex barks.
myCat.speak(); // Whiskers meows.
```

In this example, Dog and Cat classes extend the Animal class. They override the speak method to provide behavior specific to dogs and cats.

4. Static Methods

Static methods belong to the class itself rather than to instances of the class. They are often used for utility functions.

```
class MathHelper {
  static add(a, b) {
    return a + b;
  }

  static multiply(a, b) {
    return a * b;
  }
}
console.log(MathHelper.add(5, 3)); // 8
console.log(MathHelper.multiply(4, 6)); // 24
```

Here, add and multiply are static methods of the MathHelper class. You call them directly on the class itself, not on an instance.

Summary

- Constructor: Initializes new objects.
- Methods: Define behaviors.
- Inheritance: Allows creating a new class that extends an existing one.
- Static Methods: Belong to the class itself, not instances.

These examples should give you a solid understanding of JavaScript classes and how they can be used in various real-world scenarios.

Chapter 15

JavaScript Async

In JavaScript, “asynchronous” refers to operations that do not block the execution of other code while waiting for a task to complete. This is essential for improving performance and responsiveness, especially in web applications where tasks like network requests, file reading, or timers need to run in the background without freezing the user interface.

Handling asynchronous operations—such as data fetching, file reading, or timers—can be done in several ways. The primary methods are Callbacks, Promises, and Async/Await. Each has its use cases, benefits, and potential drawbacks. Let’s explore them with explanations and examples.

1. Callbacks

Explanation:

A callback is a function passed as an argument to another function and is executed after the first function completes. This pattern is useful for handling asynchronous operations.

Example:

```
// Simulate an asynchronous operation
function fetchData(callback) {
  setTimeout(() => {
    const data = “Data from server”;
    callback(null, data); // First argument is error, second is result
  }, 1000);
}
// Usage of the callback
fetchData((error, data) => {
  if (error) {
    console.error(“Error fetching data:”, error);
    return;
  }
  console.log(“Received data:”, data);
});
```

Use Cases:

- Simple asynchronous tasks.
- When working with older code or libraries that use callback-based APIs.

Drawbacks:

- Can lead to “callback hell” or “pyramid of doom” if you have multiple nested callbacks.
- Error handling can be cumbersome with nested callbacks.

2. Promises

Explanation:

A Promise is an object representing the eventual completion (or failure) of an asynchronous operation. It provides a way to attach callbacks for success or failure without nesting them.

Example:

```
// Simulate an asynchronous operation
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = “Data from server”;
      resolve(data); // Resolve with result
    }, 1000);
  });
}

// Usage of the promise
fetchData()
  .then(data => {
    console.log(“Received data:”, data);
  })
  .catch(error => {
    console.error(“Error fetching data:”, error);
  });
```

Use Cases:

- When you need better error handling and chaining of asynchronous operations.
- Modern APIs and libraries often use Promises.

Benefits:

- Avoids callback hell by allowing chaining with **.then()**.
- Provides a more readable structure for asynchronous flows.

3. Async/Await

Explanation:

Async/Await is syntactic sugar built on top of Promises. It allows you to write asynchronous code that looks synchronous, improving readability and maintainability.

Example:

```
// Simulate an asynchronous operation
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const data = "Data from server";
      resolve(data); // Resolve with result
    }, 1000);
  });
}

// Using async/await
async function getData() {
  try {
    const data = await fetchData(); // Wait for the promise to resolve
    console.log("Received data:", data);
  } catch (error) {
    console.error("Error fetching data:", error);
  }
}
```

```
// Call the async function
getData();
```

Use Cases:

- When you want to write asynchronous code in a synchronous style.
- Improves readability, especially with multiple asynchronous operations.
- Simplifies error handling using `**try...catch**`.

Benefits:

- Makes asynchronous code easier to read and maintain.
- Allows for more intuitive error handling compared to Promises alone.

Summary

- Callbacks are simple but can lead to deeply nested code and are less readable with multiple async operations.
- Promises provide a cleaner way to handle asynchronous tasks with chaining and better error handling.
- Async/Await simplifies writing asynchronous code, making it look more like synchronous code and improving readability and error handling.

Each approach has its place, and the choice often depends on the complexity of the code and personal or team preference.

Chapter 16

JavaScript Best Practices

Writing modular and reusable code in JavaScript is essential for maintaining a clean, maintainable codebase, especially as your project grows. Here's a guide to help you achieve this:

1. Use Functions and Classes

Functions:

- **Single Responsibility Principle:** Each function should do one thing and do it well. This makes it easier to test and reuse.
- **Pure Functions:** Aim to write pure functions, which are functions where the output is solely determined by the input, with no side effects. This makes them more predictable and reusable.

```
function calculateArea(radius) {
  return Math.PI * radius * radius;
}
```

Classes:

- **Encapsulation:** Use classes to bundle data and methods that operate on that data. This encapsulates functionality and makes your code more organized.

```
class Circle {
  constructor(radius) {
    this.radius = radius;
  }

  getArea() {
    return Math.PI * this.radius * this.radius;
  }
}
```

2. Modularization

Modules:

- **ES6 Modules:** Use `**import**` and `**export**` to create modular code. This allows you to separate concerns and reuse code across different files.

Exporting and Importing:

```
// utils.js
export function calculateArea(radius) {
  return Math.PI * radius * radius;
}

// main.js
import { calculateArea } from './utils.js';
console.log(calculateArea(5));
```

- **CommonJS:** For Node.js environments, use `**module.exports**` and `**require**`.

CommonJS Example:

```
// utils.js
function calculateArea(radius) {
  return Math.PI * radius * radius;
}

module.exports = { calculateArea };

// main.js
const { calculateArea } = require('./utils');

console.log(calculateArea(5));
```

3. Encapsulation and Abstraction

- **Encapsulation:** Hide implementation details and expose only what is necessary. This is typically achieved using classes or modules.
- **Abstraction:** Create abstract interfaces or classes that define a contract for what the classes should do, without specifying how they do it.

```
class Shape {
  getArea() {
    throw new Error('Method not implemented.');
  }
}

class Circle extends Shape {
  constructor(radius) {
    super();
    this.radius = radius;
  }

  getArea() {
    return Math.PI * this.radius * this.radius;
  }
}
```

4. Higher-Order Functions

- **Function Composition**: Create higher-order functions that take other functions as arguments or return functions. This promotes reusability and composability.

```
function withLogging(fn) {
  return function(...args) {
    console.log(`Calling function with args: ${args}`);
    return fn(...args);
  };
}
const add = (a, b) => a + b;
```

```
const addWithLogging = withLogging(add);

console.log(addWithLogging(1, 2));  // Logs: Calling function with args: 1,2
```

5. Error Handling and Validation

- **Validation Functions:** Write reusable validation functions to check input validity.

```
function isPositiveNumber(value) {
      return typeof value === 'number' && value > 0;
}
```

- **Error Handling:** Handle errors gracefully and create reusable error-handling functions or classes.

```
function handleError(error) {
      console.error(`An error occurred: ${error.message}`);
}
```

6. Configuration and Constants

- **Configuration Files:** Use configuration files to store settings that might change, keeping your code flexible.

```
// config.js
export const config = {
  apiUrl: 'https://api.example.com',
  timeout: 5000
};

// main.js
import { config } from './config.js';
```

- **Constants:** Use constants to define fixed values that are reused throughout your code.

```
const MAX_RETRIES = 3;
```

7. Documentation and Testing

- **Documentation:** Document your functions, classes, and modules to explain their purpose and usage. This is essential for reuse.

```
/**
 * Calculates the area of a circle.
 * @param {number} radius - The radius of the circle.
 * @returns {number} The area of the circle.
 */
function calculateArea(radius) {
  return Math.PI * radius * radius;
}
```

- **Testing:** Write unit tests to verify that your functions and classes work as expected. This makes them more reliable and easier to reuse.

```
import { calculateArea } from './utils.js';

test('calculates area of circle', () => {
  expect(calculateArea(5)).toBeCloseTo(78.54);
});
```

By following these practices, you'll create modular and reusable code that is easier to maintain and extend.

Best Practices in JavaScript

Adhering to best practices in JavaScript helps ensure your code is reliable, maintainable, and efficient. Here's a comprehensive guide covering key best practices, including what to do and what to avoid:

1. Use `const` and `let` Instead of `var`

Do:

- Prefer `**const**` for variables that don't change.
- Use `**let**` for variables that need to be reassigned.
- Avoid `**var**` as it has function scope and can lead to unexpected behavior.

Example:

```
const PI = 3.14; // Constant value
let counter = 0; // Variable that can be reassigned
```

Avoid:

- Using `**var**` for variable declarations because it can lead to issues with variable scoping.

2. Use Strict Mode

Do:

- Enable strict mode to catch common coding errors and improve performance.

Example:

```
"use strict";
function myFunction() {
  // Code in strict mode
}
```

Avoid:

- Writing code without strict mode as it may silently fail in some cases.

3. Write Modular and Reusable Code

Do:

- Break code into small, reusable functions and modules.
- Use ES6 modules (`**import**` and `**export**`) for better modularization.

Example:

```
// module.js
export function greet(name) {
  return `Hello, ${name}!`;
}

// app.js
import { greet } from './module.js';
console.log(greet('World'));
```

Avoid:

- Writing monolithic code that is hard to understand and maintain.

4. Handle Errors Gracefully

Do:

- Use `**try...catch**` for error handling in synchronous code.
- For asynchronous code, use `**.catch()**` with Promises and `**try...catch**` with `**async/await**`.

Example:

```
// Synchronous
try {
  let result = riskyFunction();
} catch (error) {
  console.error("Error:", error);
}
// Asynchronous
fetchData()
  .then(result => console.log(result))
  .catch(error => console.error("Error:", error));
```

Avoid:

- Ignoring error handling or not handling possible errors, leading to unhandled rejections and crashes.

5. Use Descriptive Variable and Function Names

Do:

- Choose meaningful and descriptive names for variables and functions to enhance code readability.

Example:

```
function calculateTotalPrice(items) {
  // Function to calculate the total price of items
}
```

Avoid:

- Using single-letter or unclear names, like `**a**`, `**b**`, or `**temp**`, which make the code hard to understand.

6. Keep Code DRY (Don't Repeat Yourself)

Do:

- Avoid code duplication by creating reusable functions or components.

Example:

```
function calculateArea(radius) {
  return Math.PI * radius * radius;
}
```

Avoid:

- Copying and pasting code with slight modifications, which makes maintenance difficult.

7. Optimize for Performance

Do:

- Avoid unnecessary computations and DOM manipulations.
- Use efficient algorithms and data structures.

Example:

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((acc, num) => acc + num, 0);
```

Avoid:

- Inefficient code that causes performance issues, such as excessive DOM manipulation or inefficient loops.

8. Follow a Consistent Coding Style

Do:

- Use a consistent coding style and adhere to a style guide, like Airbnb's or Google's JavaScript style guide.
- Utilize linting tools (e.g., ESLint) and formatters (e.g., Prettier) to enforce style consistency.

Example:

```
const greet = (name) => {
  return `Hello, ${name}!`;
};
```

Avoid:

- Mixing coding styles and formatting, which makes the codebase inconsistent and harder to read.

9. Avoid Global Variables

Do:

- Limit the use of global variables by encapsulating code within functions or modules.

Example:

```
(function() {
  const localVariable = "I am local";
  console.log(localVariable);
})();
```

Avoid:

- Polluting the global namespace with unnecessary global variables, which can lead to conflicts and bugs.

10. Use Modern JavaScript Features

Do:

- Use ES6+ features such as arrow functions, template literals, destructuring, and async/await for cleaner and more concise code.

Example:

```
const add = (a, b) => a + b;
const user = { name: 'Alice', age: 25 };
const { name, age } = user;
```

Avoid:

- Using outdated syntax and features that may lead to verbose or less efficient code.

11. Document Your Code

Do:

- Write clear comments and documentation to explain complex logic or important parts of the code.

Avoid:

- Neglecting to document important parts of the code, which can make it harder for others (or yourself) to understand later.

Summary

By following these best practices, you can write cleaner, more maintainable, and efficient JavaScript code. Avoiding common pitfalls and adhering to these guidelines will help you develop better applications and collaborate effectively with other developers.

What to Avoid

1. Avoid Global Variables:

- Global variables can lead to unexpected side effects and conflicts. Encapsulate your code in functions or modules.

2. Avoid Using `eval()`:

- `**eval()**` can execute arbitrary code and poses a security risk. Use alternatives like JSON parsing if you need to handle data.

3. Avoid `with` Statement:

- The `**with**` statement can make code unpredictable and harder to debug. It's best to avoid it.

4. Avoid Magic Numbers:

- Replace hard-coded values with named constants. This improves readability and makes your code easier to update.

5. Avoid Deeply Nested Code:

- Excessive nesting can make code hard to read and maintain. Refactor deeply nested code into smaller functions.

6. Avoid Synchronous XHR:

- Synchronous XMLHttpRequests can freeze the UI and affect performance. Use asynchronous requests instead (with `fetch` or `XMLHttpRequest`).

7. Avoid Overusing Frameworks and Libraries:

- Only use frameworks or libraries when they provide significant value. Overusing them can bloat your application.

8. Avoid Modifying Built-in Prototypes:

- Extending native objects (like `Array.prototype` or `Object.prototype`) can lead to conflicts and unexpected behavior.

9. Avoid Relying on Browser-Specific Features:

- Ensure cross-browser compatibility by using feature detection or polyfills. Avoid relying on non-standard features.

10. Avoid Inefficient DOM Manipulations:

- Minimize direct DOM manipulations and batch updates where possible to improve performance.

11. Avoid Using Deprecated APIs:

- Stay updated with current standards and avoid using deprecated APIs. This helps in maintaining compatibility with future versions of browsers.

By following these best practices, you can write JavaScript code that is efficient, maintainable, and easier to understand.

Chapter 17

Interview Questions and Answers

Here are some common JavaScript interview questions along with their answers. These cover various aspects of the language, from basic to advanced topics.

Basic Questions

1. What are the different data types in JavaScript?

Answer:

JavaScript has several data types:

- Primitive Types:

- `**Number**`: Represents both integer and floating-point numbers.
- `**String**`: Represents text.
- `**Boolean**`: Represents `true` or `false`.
- `**Undefined**`: Represents a variable that has been declared but not assigned a value.
- `**Null**`: Represents a deliberate non-value.
- `**Symbol**`: Represents a unique identifier (introduced in ES6).
- `**BigInt**`: Represents integers with arbitrary precision (introduced in ES11).

- Object Types:

- `**Object**`: General-purpose object.
- `**Array**`: Ordered collection of values.
- `**Function**`: A callable object.
- `**Date**`: Represents dates and times.
- `**RegExp**`: Represents regular expressions.

2. What is the difference between `==` and `===`?

Answer:

- `**==**` (Equality Operator) performs type coercion before comparison. It checks for value

equality after converting the operands to the same type.

```
5 == '5'; // true
```

- `===` (Strict Equality Operator) checks for both value and type equality without performing type coercion.

```
5 === '5'; // false
```

3. What is a closure in JavaScript?

Answer:

A closure is a feature where a function retains access to its lexical scope even when the function is executed outside that scope. Closures are created whenever a function is defined inside another function.

```
function outerFunction() {
    let outerVariable = 'I am outside!';
    function innerFunction() {
        console.log(outerVariable); // innerFunction has access to outerVariable
    }
    return innerFunction;
}
const closureFunction = outerFunction();
closureFunction(); // Output: 'I am outside!'
```

Intermediate Questions

4. Explain event delegation in JavaScript.

Answer:

Event delegation is a technique where a single event listener is added to a parent element to handle events for multiple child elements. Instead of adding individual event listeners to each child, the parent element listens for events on its descendants, leveraging event bubbling.

```
document.getElementById('parent').addEventListener('click', function(event) {
    if (event.target && event.target.matches('button.child')) {
        console.log('Child button clicked!');
    }
});
```

5. What is the purpose of `bind()`, `call()`, and `apply()` methods in JavaScript?

Answer:

These methods are used to control the value of `**this**` in function calls.

- `**bind()**`: Creates a new function with a bound `**this**` value and optional arguments. The bound function can be invoked later.

```
function greet() {
   console.log(`Hello, ${this.name}`);
}
const person = { name: 'Sujay' };
const greetPerson = greet.bind(person);
greetPerson(); // Output: 'Hello, Sujay'
```

- `**call()**`: Calls a function with a specified `**this**` value and arguments provided individually.

```
function greet(greeting) {
   console.log(`${greeting}, ${this.name}`);
}
const person = { name: 'Maya' };
greet.call(person, 'Hi'); // Output: 'Hi, Maya'
```

- `**apply()**`: Similar to `**call()**`, but arguments are passed as an array.

```
function greet(greeting, punctuation) {
   console.log(`${greeting}, ${this.name}${punctuation}`);
}
const person = { name: 'Charlie' };
greet.apply(person, ['Hello', '!']); // Output: 'Hello, Charlie!'
```

6. What are Promises in JavaScript?

Answer:

Promises represent the eventual completion (or failure) of an asynchronous operation and its resulting value. A Promise is an object that may produce a single value in the future: either a resolved value or a reason for failure.

```
const promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve('Success!'), 1000);
});

promise.then(result => console.log(result)) // Output: 'Success!'
       .catch(error => console.error(error));
```

Advanced Questions

7. Explain the concept of prototypal inheritance in JavaScript.

Answer:

JavaScript uses prototypal inheritance, where objects inherit properties and methods from other objects. Each object has a prototype object, and objects can access properties and methods of their prototype.

```
const animal = {
    eats: true
};

const rabbit = Object.create(animal);
console.log(rabbit.eats); // Output: true
```

8. What are JavaScript Generators?

Answer:

Generators are functions that can be paused and resumed. They use the `function*` syntax and yield values using the `yield` keyword. Generators return an iterator object.

```
function* countUpTo(max) {
    let count = 1;
    while (count <= max) {
        yield count++;
    }
}

const counter = countUpTo(3);
```

```
console.log(counter.next().value); // Output: 1
console.log(counter.next().value); // Output: 2
console.log(counter.next().value); // Output: 3
```

9. What is the difference between `null` and `undefined`?

Answer:

- `**null**` is a deliberate assignment of an empty or non-existent value.

```
let emptyValue = null;
```

- `**undefined**` means a variable has been declared but not assigned a value.

```
let notAssigned;
console.log(notAssigned); // Output: undefined
```

10. What is the event loop in JavaScript?

Answer:

The event loop is a mechanism that handles asynchronous operations in JavaScript. It continuously checks the call stack and the message queue. When the call stack is empty, it processes messages from the queue. This allows JavaScript to perform non-blocking I/O operations and manage asynchronous code execution.

```
console.log('Start');
setTimeout(() => {
    console.log('Timeout');
}, 0);
console.log('End');
// Output:
//   Start
//   End
//   Timeout
```

These questions cover a range of JavaScript topics and can help evaluate a developer's understanding of the language's features and best practices.

Here are a few more JavaScript interview questions and their answers:

11. What is the difference between `apply()` and `call()` methods in JavaScript?

Answer:

Both `**apply()**` and `**call()**` are methods used to invoke a function with a specific `this` value and arguments. The difference lies in how they handle arguments:

- `**call()**`: Accepts arguments individually.

```
function greet(greeting, name) {
   console.log(`${greeting}, ${name}`);
}
greet.call(null, 'Hello', 'World'); // Output: 'Hello, World'
```

- `**apply()**`: Accepts arguments as an array.

```
function greet(greeting, name) {
   console.log(`${greeting}, ${name}`);
}
greet.apply(null, ['Hello', 'World']); // Output: 'Hello, World'
```

12. How does the `this` keyword work in JavaScript?

Answer:

The `**this**` keyword refers to the context in which a function is executed. Its value depends on how the function is called:

- Global context: In the global execution context, `this` refers to the global object (window in browsers).
- Object method: Inside an object method, `this` refers to the object itself.

```
const person = {
   name: 'Alice',
   greet() {
       console.log(this.name);
   }
};
person.greet(); // Output: 'Alice'
```

- **Constructor function**: In a constructor function, `**this**` refers to the new instance being created.
- **Arrow functions**: Arrow functions do not have their own `**this**`; they inherit `**this**` from their surrounding lexical context.

13. What is the purpose of `Object.freeze()`?

Answer:

`**Object.freeze()**` is used to make an object immutable. It prevents new properties from being added, existing properties from being removed, and existing properties from being modified.

```
const obj = { name: 'Vijay' };
Object.freeze(obj);
obj.name = 'Jay'; // No effect
console.log(obj.name); // Output: 'Vijay'
```

14. What is the `new` keyword and how does it work?

Answer:

The `**new**` keyword is used to create an instance of a constructor function. It performs the following steps:

1. Creates a new object.
2. Sets the prototype of the new object to the prototype of the constructor function.
3. Executes the constructor function with `this` bound to the new object.
4. Returns the new object if the constructor does not return another object.

```
function Person(name) {
    this.name = name;
}
const person = new Person('Alice');
console.log(person.name); // Output: 'Alice'
```

15. What is event bubbling and how does it differ from event capturing?

Answer:

- **Event Bubbling**: Events start from the target element and bubble up to the root. This is the default behavior in most browsers.

```
document.getElementById('child').addEventListener('click', () => {
    console.log('Child clicked');
}, false); // Event bubbling phase
```

- **Event Capturing**: Events start from the root and travel down to the target element. This is less commonly used and requires the third parameter of `**addEventListener**` to be set to `**true**`.

```
document.getElementById('parent').addEventListener('click', () => {
    console.log('Parent clicked');
}, true); // Event capturing phase
```

16. What is a JavaScript module, and how do you use them?

Answer:

JavaScript modules are a way to encapsulate and organize code into reusable pieces. Modules can export values (functions, objects, etc.) and import values from other modules.

- Exporting:

```
// module.js
export const greet = (name) => `Hello, ${name}`;
```

- Importing:

```
// app.js
import { greet } from './module.js';
console.log(greet('Alice')); // Output: 'Hello, Alice'
```

Modules are supported natively in modern JavaScript environments (e.g., ES6 modules).

17. Explain the concept of "hoisting" in JavaScript.

Answer:

Hoisting is a behavior in JavaScript where variable and function declarations are moved to the top of their containing scope during compilation. This means that functions and variables can be used before they are declared.

```
console.log(x); // Output: undefined
var x = 5;

greet(); // Output: 'Hello'
function greet() {
    console.log('Hello');
}
```

Note that only the declarations are hoisted, not the initializations.

18. What is the `typeof` operator and what are its limitations?

Answer:

The `**typeof**` operator returns a string indicating the type of the operand.

```
console.log(typeof 'Hello'); // Output: 'string'
console.log(typeof 42); // Output: 'number'
console.log(typeof true); // Output: 'boolean'
console.log(typeof {}); // Output: 'object'
console.log(typeof []); // Output: 'object'
console.log(typeof null); // Output: 'object' (historical bug)
console.log(typeof function() {}); // Output: 'function'
```

The `**typeof**` operator has some limitations, such as incorrectly identifying arrays and `**null**` as objects.

19. What are template literals and how are they different from regular strings?

Answer:

Template literals, introduced in ES6, allow for multi-line strings and embedded expressions. They use backticks (``) instead of single or double quotes.

```
const name = 'Vijay';
const greeting = `Hello, ${name}!`; // Template literal with embedded expression
console.log(greeting); // Output: 'Hello, Vijay!'
```

Unlike regular strings, template literals can span multiple lines and embed expressions using `**${}**`.

20. What is the purpose of the `with` statement, and why is it considered bad practice?

Answer:

The `**with**` statement extends the scope chain for a block of code, allowing for more concise access to properties of an object. However, it is considered bad practice due to the confusion it can cause with variable scope and potential performance issues.

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

It is deprecated in strict mode and should be avoided in modern JavaScript.

These additional questions and answers cover a broader range of topics and help assess a developer's knowledge of JavaScript in more depth.

Thank You !