JavaScript Fundamentals

- **Primitives**

  Is a data that is not an object and has no methods or properties. They are immutable; that is, they cannot be altered. Primitives have no methods but still behave as if they do. When properties are accessed on primitives, Javascript auto-boxes the value into a wrapper object and accesses the property on that object instead. ***"foo".includes("f");***

- **Scopes and Hoisting**

  Scopes refers to the current context of execution, which determines the accessibility of variables.

  Types of Scopes are:

  1) **Global Scope:**

     They are declared outside the function or the block. Accessible anywhere in the code.

     ```
     let globalVar = "Global";
     Function example() {
         console.log(globalVar)
     }
     example();
     console.log(globalVar);
     ```

  2) **Function Scope:**

     Variables declared with var, let or const inside a function are scoped to that function. Not accessible outside the function.

     ```
     Function example(){
         Let functionVar = "Inside the function";
         console.log(functionVar);
     }
     example();
     console.log(functionVar); // Error: functionVar is not defined
     ```

  3) **Block Scope:**

     Variables declared with let and const inside a block {} are limited to that block. Var does not respect block scope(it is function-scoped).

     ```
     if(true) {
         let blockVar = "I am blocked-scoped";
         console.log(blockVar);
     }
     console.log(blockVar); // Error: blockVar is not defined
     ```

  - **Hoisting**

Hoisting is javascript's default behaviour of moving declarations to the top of their scope4 before code execution. Only declarations are hoisted, not initializations.

- **Variable Hoisting:**
    1) **var:**
       Variables declared with var are hoisted and initialized with undefined
       console.log(hoistedVar); // undefined
       var hoistedVar = "I am hoisted";
       console.log(hoistedVar); // " I am hoisted"
    2) **Let and const:**
       Variables declared with let or const are hoisted but remain in the "**Temporal Dead Zone"** until they are initialized.
       console.log(hoistedLet); // Error: Cannot access 'hoistedLet' before initialization
       Let hoistedLet = ' I am not accessible before initialization';
- **Function Hoisting:**
    1) **Function Declarations:**
       Function declarations are hoisted entirely, making them accessible before their definition.
       hoistedFunction();
       hoistedFunction() {
       console.log("I am hoisted");
       }
    2) **Function Expressions:**

Function expressions (including arrow functions) are treated like variables.

console.log(hoistedFunc); // undefined
var hoistedFunc = function() {
console.log("I am not hoisted");
}

```
function scopeAndHoisting() {
  console.log(a); // undefined (hoisted)
  console.log(b); // Error: Cannot access 'b' before initialization
  console.log(c); // Error: Cannot access 'c' before initialization

  var a = 10; // Function-scoped
  let b = 20; // Block-scoped
  const c = 30; // Block-scoped

  if (true) {
      var a = 40; // Overrides the outer `a`
```

```
        let b = 50; // Block scope, different from outer `b`
        const c = 60; // Block scope, different from outer `c`
        console.log(a, b, c); // 40, 50, 60
    }

    console.log(a); // 40 (var is function-scoped)
    console.log(b); // 20 (block-scoped `let`)
    console.log(c); // 30 (block-scoped `const`)
}

scopeAndHoisting();
```

- **Closures**

  A closure is a function that 'remembers' the variables from its outer scope even after the outer function has finished executing. Closures are created every time a function is defined inside another function and the inner function accesses variables from the outer function.

  1) Access to outer scope variables
  2) Encapsulations: allow you to encapsulate data, creating private variables that can't be accessed directly from the outside of the function.
  3) Persistent Data: Variables in the closure's scope persist in memory as long as the closure exists.

```
function outerFunction() {
  let outerVar = "I am from the outer scope";

  function innerFunction() {
    console.log(outerVar); // Accesses outerVar from the outer function
  }

  return innerFunction;
}

const closure = outerFunction();
closure(); // Logs: "I am from the outer scope"
```

Practical use case of Closures

1) **Data Privacy**

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

  return {
    increment() {
      count++;
      return count;
```

```
        },
        decrement() {
            count--;
            return count;
        },
        getCount() {
            return count;
        }
    };
}

const counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.increment()); // 2
console.log(counter.decrement()); // 1
console.log(counter.getCount());  // 1
```

2) **Function Factories**

```
function multiplyBy(factor) {
    return function (num) {
        return num * factor; // Uses the factor from the outer scope
    };
}

const multiplyBy2 = multiplyBy(2);
const multiplyBy3 = multiplyBy(3);

console.log(multiplyBy2(5)); // 10
console.log(multiplyBy3(5)); // 15
```

3) **Memoization**

Closures can store results of expensive function calls for performance optimization.

```
function memoize(fn) {
    const cache = {};

    return function(args) {
        if(cache[args]){
            console.log("result found in the cache");
            return cache[args];
        }
```

```
        console.log("Computing the expensive calculation");
        const result = fn(args);
        cache[args] = result;
        return result;
    }
}

const square = memoize((n) => n * n);
console.log(square(5));
console.log(square(5));
```

- **Execution Context**

  In Javascript, the execution context is an environment in which a piece of javascript code is evaluated and executed. It manages the code execution by keeping track of variables, functions and objects and determines how they interact during the runtime. Understanding execution context is key to mastering concepts like scope, closures and this.

  Types of Execution Contexts:

  1) **Global Execution Context(GEC):**

     Default context where the execution starts. It is created when the script begins to execute. Responsible for managing the global variable and functions.Associated with window object in browsers or the global object in Node.js

     ```
     var a = 10; // Part of the global execution context
     function foo() {
        console.log("Hello");
     }
     ```

  2) **Function Execution Context(FEC):**

     Created whenever a function is invoked. Each function has its own execution context. Contains information like: Args passed to the function, local variables, scope chain(access to outer scopes), value of this.

     ```
     function example() {
        let b = 20; // Local variable in the function's execution context
        console.log(b);
     }
     example();
     ```

  3) **Eval Execution Context (rarely used):**

     Created when code is executed inside eval. Not recommended for use due to performance and security issues.

  Components of an Execution Context

  An execution context consists of three main components:

1)Variable Environment:

Stores variable and function declarations. Includes let, const, and var variables. Handled hoisting for var declarations.

```
console.log(a); // undefined (hoisted)
var a = 10;
```

2) Lexical Environment

Contains the current environment's variables and references to its parent environment (scope chain). Enables access to variables declared in outer functions or the global scope.

```
function outer() {
   let outerVar = "Outer";

   function inner() {
      console.log(outerVar); // Access to outer function variable
   }
   inner();
}
outer();
```

3) this Binding:

Refers to the object that the function is currently bound to. Depends on how the function is called.

-> Global Context: this refers to the global object **window** and **global**

-> Function Context: Depends on the invocation method (e.g method call, constructor call, or bind/apply)

```
console.log(this); // In global scope, `this` refers to the global object

const obj = {
   method: function () {
      console.log(this); // Refers to `obj`
   }
};
obj.method();
```

- **Variables(let, var and const)**
- **Operators**
    1) **Arithmetic operators (+,-,*,/,%,**)**
    2) **Assignment Operators (=, +=. -=, *=, /=, %=)**
    3) **Comparison Operators (==, ===, !=, !===,>, <, <=, >=)**
    4) **Logical Operators (&&, ||, !)**
    5) **Bitwise Operators (&, |, ^ = XOR, ~=NOT, << Left Shift, >> Right Shift)**

6) **Ternary Operators(shorthand if..else) let isAdult = (age >= 18) ? “yes” : “no”;**
7) **Type Operators (typeof => returns type, instanceof => check instance)**
8) **Nullish Coalescing Operator (??) returns the right-hand value if the left hand value is null or undefined.**
9) **Optional Chaining Operator (?.) safely access deeply nested properties without throwing an error (user.address?.street)**
10) **Unary Operators (single operand) (+ => +”5” converts to number, - negates a number, ++ increment, – decrement)**
11) **Spread and Rest Operators (...)**
    - **Spread: Expands elements.**
    - **Rest: Collects elements into an array.**

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

// Rest
function sum(...nums) {
   return nums.reduce((a, b) => a + b);
}
console.log(sum(1, 2, 3)); // 6
```

- **Type Conversion**

In JavaScript, type conversion refers to changing a value from one data type to another. Conversions can happen automatically (implicit) or explicitly (manual).

1) Implicit Type Conversion (Type Coercion)
2) Explicit Type Conversion (Type Casting)

```
let num = 123;
let str = String(num);
console.log(typeof str); // "string"

let num = 123;
let str = num.toString();
console.log(typeof str); // "string"

let str = "123";
let num = Number(str);
console.log(typeof num); // "number"

let str = "123.45";
let intNum = parseInt(str);  // 123
let floatNum = parseFloat(str); // 123.45
```

```
let str = "123";
let num = +str;
console.log(typeof num); // "number"
```

```
let str = "hello";
let isTrue = Boolean(str); // true
console.log(isTrue);
```

- **Array and its methods**

An **array** is a special object used to store multiple values in a single variable. Each value in an array has an **index** (starting from `0`) and can be of any data type.

```
let numbers = [1, 2, 3, 4];       // Array of numbers
let fruits = ["Apple", "Banana"]; // Array of strings
let mixed = [1, "hello", true];   // Array of mixed types
let empty = [];                   // Empty array
```

Array Methods

JavaScript provides a variety of built-in methods to work with arrays. Here's a categorized explanation of the most useful methods:

1) **Adding/Removing Elements**

`push()`: Adds one or more elements to the end of the array.

```
let fruits = ["Apple", "Banana"];
fruits.push("Orange");
console.log(fruits); // ["Apple", "Banana", "Orange"]
```

`pop()`: Removes the last element from the array.

```
fruits.pop();
 console.log(fruits); // ["Apple", "Banana"]
```

`unshift()`: Adds one or more elements to the beginning of the array.

```
let fruits = ["Banana"];
 fruits.unshift("Apple");
console.log(fruits); // ["Apple", "Banana"]
```

`shift()`: *Removes the first element from the array.*

```
let fruits = ["Apple", "Banana"];
fruits.shift();
console.log(fruits); // ["Banana"]
```

2) **Accessing/Checking Elements**

`indexOf()`: **Returns the index of the first occurrence of a value.**

```
let fruits = ["Apple", "Banana", "Orange"];
console.log(fruits.indexOf("Banana")); // 1
```

`includes()`: **Checks if a value exists in the array.**

```
let fruits = ["Apple", "Banana"];
console.log(fruits.includes("Orange")); // false
```

3) **Iterating Through Arrays**

`forEach()`: **Executes a function for each array element.**

```
let fruits = ["Apple", "Banana", "Orange"];
 fruits.forEach(fruit => console.log(fruit));
```

`map()`: *Creates a new array by transforming each element.*

```
let numbers = [1, 2, 3];
let squares = numbers.map(num => num * num);
console.log(squares); // [1, 4, 9]
```

4) **Filtering/Reducing**

`filter()`: **Returns a new array with elements that pass a test.**

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

`reduce()`: **Reduces the array to a single value.**

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

5) **Modifying/Reorganizing Arrays**

`splice()`: **Adds, removes, or replaces elements in the array.**

```
let fruits = ["Apple", "Banana", "Orange"];
fruits.splice(1, 1, "Grapes"); // Remove 1 element at index 1 and add "Grapes"
console.log(fruits); // ["Apple", "Grapes", "Orange"]
```

`slice()`: **Returns a shallow copy of a portion of the array.**

```
let fruits = ["Apple", "Banana", "Orange"];
let citrus = fruits.slice(1, 3); // From index 1 to index 3 (not included)
console.log(citrus); // ["Banana", "Orange"]
```

`reverse()`: **Reverses the order of the elements.**

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

`sort()`: **Sorts the elements in ascending (default) order.**

```
let fruits = ["Banana", "Apple", "Orange"];
fruits.sort();
console.log(fruits); // ["Apple", "Banana", "Orange"]

let numbers = [10, 5, 20];
numbers.sort((a, b) => a - b); // Numeric sort
console.log(numbers); // [5, 10, 20]
```

6) **Joining/Converting Arrays**

`join()`: **Joins all elements into a string.**

```
let fruits = ["Apple", "Banana", "Orange"];
let result = fruits.join(", ");
console.log(result); // "Apple, Banana, Orange"
```

`toString()`: **Converts an array to a string.**

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

7) **Searching**

`find()`: **Returns the first element that satisfies the condition.**

```
let numbers = [1, 2, 3, 4];
let result = numbers.find(num => num > 2);
console.log(result); // 3
```

`findIndex()`: **Returns the index of the first element that satisfies the condition.**

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

8) **Combining/Flattening Arrays**

`concat()`: **Combines two or more arrays.**

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

`flat()`: **Flattens nested arrays.**

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

- **Object and its method**

An object is a collection of key-value pairs where keys (or properties) are strings (or Symbols), and values can be of any type (primitives or other objects). Objects are one of the fundamental building blocks in JavaScript.

Creating Objects

1. **Object Literal Syntax**:

```
let person = {
  name: "John",
  age: 30,
  greet: function () {
    console.log(`Hello, my name is ${this.name}`);
  }
};
```

2 **Using the `Object` Constructor**:

```
let person = new Object();
person.name = "John";
person.age = 30;
```

3 **Using Object.create()**:

```
let proto = { greet() { console.log("Hello!"); } };
let person = Object.create(proto);
person.name = "John";
```

Accessing object properties

1) *Dot Notation*

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

2) *Bracket Notation*

```
console.log(person["age"]); // 30
```

3) *Dynamic Property Access*

```
let key = "name";
console.log(person[key]); // "John"
```

`Object.assign(target, ...sources)`: *Copies properties from one or more source objects to a target object.*

```
let target = { a: 1 };
let source = { b: 2, c: 3 };
Object.assign(target, source);
console.log(target); // { a: 1, b: 2, c: 3 }
```

2. Property Management

1. `Object.keys(obj)`: *Returns an array of the object's property keys.*

```
let obj = { name: "John", age: 30 };
console.log(Object.keys(obj)); // ["name", "age"]
```

`Object.values(obj)`: *Returns an array of the object's property values.*

```
console.log(Object.values(obj)); // ["John", 30]
```

`2. Object.entries(obj)`: Returns an array of `[key, value]` pairs.
console.log(Object.entries(obj)); // [["name", "John"], ["age", 30]]

`3. Object.freeze(obj)`: Freezes the object, making it immutable.
let obj = { name: "John" };
Object.freeze(obj);
obj.name = "Doe"; // No effect
console.log(obj.name); // "John"

`4. Object.seal(obj)`: Seals the object, preventing new properties from being added or existing properties from being deleted, but allows modification of existing properties.
let obj = { name: "John" };
Object.seal(obj);
obj.age = 30; // No effect
obj.name = "Doe"; // Allowed
console.log(obj); // { name: "Doe" }

3. Prototype Management
`Object.getPrototypeOf(obj)`: Returns the prototype of the object.
let obj = Object.create({ greet() { console.log("Hi"); } });
console.log(Object.getPrototypeOf(obj)); // { greet: [Function: greet] }

`Object.setPrototypeOf(obj, proto)`: Sets the prototype of the object.
let proto = { greet() { console.log("Hello"); } };
let obj = {};
Object.setPrototypeOf(obj, proto);
obj.greet(); // "Hello"

4. Checking Properties
`hasOwnProperty(prop)`: Checks if the object has the property as its own (not inherited).
let obj = { name: "John" };
console.log(obj.hasOwnProperty("name")); // true
console.log(obj.hasOwnProperty("toString")); // false (inherited)

`in` Operator: Checks if the property exists (including inherited).
console.log("name" in obj); // true
console.log("toString" in obj); // true

5. Iterating Over Properties
`for...in` Loop: Iterates over all enumerable properties (including inherited ones).
```
for (let key in obj) {
   console.log(key); // "name"
}
```

`Object.entries()` with `for...of`: Combines keys and values for iteration.

```
for (let [key, value] of Object.entries(obj)) {
   console.log(`${key}: ${value}`); // "name: John"
}
```

– Functions in JavaScript

A function is a reusable block of code designed to perform a specific task. Functions allow you to encapsulate logic and reuse it as needed.

1. Declaring Functions

a. Function Declaration

```
function add(a, b) {
   return a + b;
}
console.log(add(2, 3)); // 5
```

b. Function Expression

```
const multiply = function (a, b) {
   return a * b;
};
console.log(multiply(2, 3)); // 6
```

c. Immediately Invoked Function Expression (IIFE)

```
(function () {
   console.log("IIFE executed!");
})();
```

2. Arrow Functions

Arrow functions are a concise way to write functions introduced in ES6. They are always anonymous and have different behavior for `this`.

```
const add = (a, b) => a + b; // Single-line return
console.log(add(2, 3)); // 5
```

Key Features of Arrow Functions

1. No `this` Binding: Arrow functions do not have their own `this`. They inherit `this` from the surrounding context.
2. Simpler Syntax: Compact syntax, especially for small functions.
3. No `arguments` Object: Arrow functions do not have the `arguments` object.

Normal functions create their own `this` based on how they are called. Arrow functions do not.

```
const obj = {
    value: 42,
    normalFunc: function () {
        console.log(this.value); // Refers to `obj`
    },
    arrowFunc: () => {
        console.log(this.value); // Refers to outer scope, not `obj`
    },
};
obj.normalFunc(); // 42
obj.arrowFunc(); // undefined (or outer `this` context)
```

Try-Catch and Error Handling in JavaScript

Error handling in JavaScript ensures that your code can gracefully recover from unexpected issues instead of crashing. The `try...catch` block is the primary mechanism for handling errors.

```
try {
    let result = 10 / 0; // No error here
    let undefinedVar = x; // Throws ReferenceError
    console.log("This will not execute");
} catch (error) {
    console.log("An error occurred:", error.message); // An error occurred: x is not defined
} finally {
    console.log("Execution finished."); // Always executes
}
```

Throwing Custom Errors

You can create and throw custom errors using the `throw` keyword.

```
function divide(a, b) {
    if (b === 0) {
        throw new Error("Division by zero is not allowed.");
    }
    return a / b;
}
```

```
try {
    console.log(divide(10, 0));
} catch (error) {
    console.error("Error:", error.message); // Error: Division by zero is not allowed.
}
```

JavaScript Strict Mode

Strict mode in JavaScript is a way to enforce stricter parsing and error handling in your code. It prevents the usage of certain unsafe features and helps you write cleaner and more secure code.

Introduced in ECMAScript 5 (ES5), strict mode can be applied globally or to specific functions.

How to Enable Strict Mode

1. *Globally Add `"use strict";` at the top of a script file to enforce strict mode for the entire script.*

```
"use strict";

x = 10; // Error: x is not defined
console.log(x);
```

2. *Locally (Inside Functions) Add `"use strict";` at the top of a function to enforce strict mode only within that function.*

```
function myFunction() {
    "use strict";
    y = 10; // Error: y is not defined
}
myFunction();
```

Why Use Strict Mode?

1. *Better Error Handling Helps developers catch potential bugs early.*
2. *Enhanced Performance Some JavaScript engines can optimize code better in strict mode.*
3. *Prepares for Future JavaScript Restricts usage of features that may lead to issues with newer versions of JavaScript.*
4. *Improves Security Helps avoid unintentional global variables and accidental overwrites.*

`setTimeout()` and `setInterval()` in JavaScript

Both `setTimeout()` and `setInterval()` are used to schedule tasks to run after a certain amount of time. However, they differ in how they execute those tasks:

1. setTimeout()

Purpose: Executes a function once after a specified delay.

2. setInterval()

Purpose: Executes a function repeatedly with a fixed time delay between each call.

Stopping `setTimeout()` and `setInterval()`

1. **Stopping `setTimeout()`:**
   `setTimeout()` returns a timeout ID, which you can use with `clearTimeout()` to cancel the execution of the scheduled function before the time expires.

```
let timeoutId = setTimeout(() => {
    console.log("This won't be executed.");
}, 5000);

clearTimeout(timeoutId); // Cancels the timeout before it runs
```

Stopping setInterval():

setInterval() returns an interval ID, which you can use with clearInterval() to stop the repeated execution of the function.

```
let intervalId = setInterval(() => {
    console.log("This will be logged every second.");
}, 1000);

setTimeout(() => {
    clearInterval(intervalId); // Stops the interval after 5 seconds
    console.log("Interval stopped.");
}, 5000);
```

Creating a Class

To define a class, you use the `class` keyword followed by the class name and its constructor.

```
class Person {
   constructor(name, age) {
      this.name = name;
      this.age = age;
   }

   greet() {
      console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);
   }
}

// Creating an instance of the class
const person1 = new Person("Alice", 30);
person1.greet(); // Output: Hello, my name is Alice and I am 30 years old.
```

`constructor`: A special method used for creating and initializing an object created with a class. The `constructor` is called when a new instance of the class is created.

```
class Person {
   constructor(name, age) {
      this.name = name;
      this.age = age;
   }
}
```

Methods: Regular functions that belong to the class. They define behaviors for instances of the class.

```
class Person {
   greet() {
      console.log(`Hello, ${this.name}!`);
   }
}
```

Getter and Setter Methods

In classes, getter and setter methods allow you to control the access and modification of object properties.

Example of Getters and Setters

```
class Person {
   constructor(name, age) {
```

```
        this._name = name;
        this._age = age;
    }

    // Getter
    get name() {
        return this._name;
    }

    // Setter
    set name(newName) {
        this._name = newName;
    }

    // Getter for age
    get age() {
        return this._age;
    }

    // Setter for age
    set age(newAge) {
        if (newAge > 0) {
            this._age = newAge;
        } else {
            console.log("Age must be positive.");
        }
    }
}

const person = new Person("Alice", 25);
console.log(person.name); // Output: Alice
person.name = "Bob"; // Setter is called
console.log(person.name); // Output: Bob
person.age = -5; // Invalid age, will not be set
```

Static Methods

Static methods are called on the class itself, not on instances of the class.

Example of Static Method

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

```
    }

    static subtract(a, b) {
        return a - b;
    }
}

console.log(Calculator.add(5, 3)); // Output: 8
console.log(Calculator.subtract(5, 3)); // Output: 2
```

Class Expressions

Classes can also be created using expressions.

Example of Class Expression

```
const Person = class {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    greet() {
        console.log(`Hello, my name is ${this.name}`);
    }
};

const person = new Person("Charlie", 40);
person.greet(); // Output: Hello, my name is Charlie
```

Class Method Overriding

You can override a method in a subclass that inherits from a parent class.

```
class Animal {
    speak() {
        console.log("Animal makes a sound.");
    }
}

class Dog extends Animal {
    speak() {
        console.log("Dog barks.");
```

```
    }
}

const dog = new Dog();
dog.speak(); // Output: Dog barks.
```

Private Fields (ES2022)

ES2022 introduced the concept of private fields in classes using the # syntax. This allows encapsulation, making certain properties of the class accessible only within the class itself.

Example of Private Fields

```
class Person {
    #name; // Private field

    constructor(name) {
        this.#name = name;
    }

    getName() {
        return this.#name;
    }
}

const person = new Person("Alice");
console.log(person.getName()); // Output: Alice
console.log(person.#name); // Error: Private field '#name' must be declared in an enclosing class
```