JAVASCRIPT

JS

CONTENTS

Introduction

Variables & Data Types

Arithmetic & Comparison Operators

Conditional Statements

Loops

High Order Array Methods

OOP

Introduction

What Is JavaScript

- A scripting language for creating dynamic web pages.
- Runs in browsers and on servers (Node.js).
- One of the three core web technologies: HTML, CSS, JavaScript.
- Can manipulate web content, respond to user actions, and communicate with servers.

JavaScript Is Not Java

- Java and JavaScript are distinct programming languages despite the similarity in their names. This naming convention was largely a marketing decision by Netscape to leverage the popularity of Java at the time JavaScript was released.

- Java is a compiled, object-oriented programming language, while JavaScript is primarily an interpreted scripting language.

Variables & Data Types

Variables In JS

- Variables store data values for reuse in your programs.
- You can define data using **let**, **const**, or **var**
- let: can be reassigned; const: can’t be reassigned. While var shouldn't be used in modern JS
- Variable names can include letters, numbers, _, and $, but can’t start with a number.

Data Types In JS

- Primitive Data Types: These represent single, immutable values.
  - Number (1, -5, 3.14)
  - String ('Hello, world!')
  - Boolean (true or false)
  - Undefined
  - Null
  - BigInt (whole numbers larger than 2^53 – 1)
- Non-Primitive (Reference) Data Types: These are mutable and represent more complex data structures.
  - Object ({ firstname: 'Ahmed', lastname: 'Mohamed', age: 31 })
  - Array ([1,2,3,4,'five'])
  - Function (function greet() { return 'Hello'; })

Operators

Arithmetic Operators In JS

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Modulus)
- ** (Exponentiation)
- ++ (Increment)
- -- (Decrement)

Comparison Operators In JS

- == (Equal)
- != (Not Equal)
- === (Strictly Equal)
- !== (Strictly Not Equal)
- > (Greater Than)
- >= (Greater Than or Equal)
- < (Smaller Than)
- <= (Smaller Than or Equal)

Conditional Statements

If, else & else if

- JavaScript conditional statements allow for the execution of different code blocks based on whether specific conditions are met. These statements are fundamental for controlling program flow and enabling dynamic behavior.
- if statement: Executes a block of code if a specified condition evaluates to true.
- else statement: Used in conjunction with an if statement, it executes a block of code if the if statement's condition evaluates to false.
- else if statement: Allows for testing multiple conditions sequentially. If the preceding if or else if conditions are false, a new condition is tested.
- Ternary (Conditional) Operator: A shorthand for simple if-else statements, it evaluates a condition and returns one of two expressions based on whether the condition is true or false. (variable = (condition) ? DoThisIfTrue : ElseDoThisIfFalse;)

Switch Statement

- switch statement: Provides an alternative to lengthy if-else if chains when dealing with multiple possible values for a single expression. It evaluates an expression and executes the code block associated with the matching case.

```
switch (expression){
  case value1:
    // Code if expression matches value1
    break; // Exits the switch statement
  case value2:
    // Code if expression matches value2
    break;
  default:
    // Code if no case matches the expression
}
```

For & While Loops

For Loop

- In JavaScript, a for loop is a control flow statement that allows code to be executed repeatedly based on a condition. It is commonly used for iterating a known number of times or for processing elements within an array or other iterable objects.

- Standard for loop: This is the most common type, used when you know the exact number of iterations or need fine-grained control over the loop's progression.

```
for (let i = 0; i < 5; i++) {

    console.log(i); // Outputs: 0, 1, 2, 3, 4

}
```

For...in Loop

- This loop iterates over the enumerable string properties of an object. It is generally not recommended for iterating over arrays if the order of elements is important, as the order of iteration is not guaranteed.

```
const person = { name: "Alice", age: 30 };
for (let prop in person) {
   console.log(`${prop}: ${person[prop]}`);
}


// Outputs:
// name: Alice
// age: 30
```

For...of Loop

- This loop iterates over the values of iterable objects (like arrays, strings, Maps, Sets, etc.). It is the preferred method for iterating over array elements when you need access to the values directly.

```
const numbers = [1, 2, 3];
for (let num of numbers) {
    console.log(num);
}

// Outputs: 1, 2, 3
```

ForEach Loop

- While not a traditional for loop statement, the forEach() method is a common way to iterate over array elements. It is a higher-order function that takes a callback function to be executed for each element.

```
const fruits = ["apple", "banana", "cherry"];
fruits.forEach(function(fruit) {
    console.log(fruit);
});

// Outputs: apple, banana, cherry
```

While Loop

- A while loop in JavaScript is a control flow statement that repeatedly executes a block of code as long as a specified condition evaluates to true. This type of loop is particularly useful when the number of iterations is unknown beforehand, and the loop needs to continue until a certain condition is met.

```
let count = 0;

while (count < 5) {
console.log("Current count: " + count);
count++; // Increment count to eventually make the condition false
}

console.log("Loop finished. Final count: " + count);
```

High Order Array Methods

Map Function

- Higher-order array methods in JavaScript are built-in methods of the Array.prototype that accept functions as arguments (callback functions) to perform operations on each element of an array. These methods are fundamental to functional programming paradigms in JavaScript, enabling concise and expressive code for data manipulation.

- map(): Creates a new array by calling a provided function on every element in the calling array. It returns a new array with the transformed elements.

```
const numbers = [1, 2, 3];
const doubledNumbers = numbers.map((num) => num * 2);
console.log(doubledNumbers)

//  Output: [2, 4, 6]
```

Filter Function

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

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

// Output: [2,4]
```

Find Function

- find(): Returns the value of the first element in the provided array that satisfies the provided testing function. Otherwise, undefined is returned.

```
const numbers = [1, 2, 3, 4, 5];
const firstEven = numbers.find((num) => num % 2 === 0);
console.log(firstEven);

// Output: 2
```

OOP

What Is OOP In JS

- Object-Oriented Programming (OOP) is a way to structure programs by modeling data and behavior as objects.
- We use classes to act as blueprints for building instances (objects) of that class
- JavaScript uses prototype-based inheritance, where objects can share behavior by linking to a common prototype.
- OOP is based on four main principles: encapsulation, inheritance, polymorphism, and abstraction. These principles help organize code, improve reusability, and make software more manageable
- Classes were introduced in ES6, it provided modern, cleaner syntax for creating constructor functions with shared methods, but under the hood, they still use prototypes.

Before VS After ES6

- Before:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
Person.prototype.sayHello = function() {
  console.log(`Hello, my name is ${this.name}`);
};
```

- After:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`Hello, my name is ${this.name}`);
  }
}
```

Encapsulation

- Encapsulation means bundling data and the methods that operate on that data inside a single object, while restricting direct access to some of the object's internal details.
- It helps protect data from unintended interference and misuse by other parts of your code.
- In many OOP languages, you can mark properties as private so they can't be accessed directly – this is the essence of true encapsulation.
- Traditionally, JavaScript didn't have true private properties like Java or C++ because everything in an object was public. But you could simulate encapsulation with:
    - Naming conventions – e.g., _propertyName to signal it's "private."
    - Closures – functions inside constructors to hide data.
    - ES6+ setters and getters – control access to properties.
    - ES2022+ private class fields (with #) – true private properties!

Encapsulation (Setters & Getters)

```
class User {
  constructor(name) {
    this._name = name;                    // convention: underscore means “private”, but "not really hidden" can access directly!
  }

  get name() {
    return this._name;
  }

  set name(value) {
    if (typeof value !== 'string') {
      throw new Error('Name must be a string');
    }
    this._name = value;
  }
}

const user = new User('Ahmed');
console.log(user.name);                   // Uses getter → "Ahmed"

user.name = 'Sara';     // Uses setter
console.log(user.name);                   // Output:  "Sara"
```

Encapsulation (Setters & Getters) ES2022+

```
class Account {
  #balance;

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

  get balance() {
    return this.#balance;
  }

  deposit(amount) {
    this.#balance += amount;
  }
}
const account = new Account(1000);
console.log(account.balance);         // 1000

account.deposit(500);
console.log(account.balance);         // 1500
console.log(account.#balance);        //  SyntaxError: private field cannot be accessed
```

Inheritance

- Inheritance in JavaScript is a mechanism that allows objects to acquire properties and methods from other objects, promoting code reusability and establishing hierarchical relationships.
- Introduced in ECMAScript 2015 (ES6), this provides a more familiar syntax for object-oriented programming, resembling class-based inheritance in other languages like Java or C++.
- The class keyword defines a blueprint for creating objects, and the extends keyword is used to establish inheritance between classes.
- The super() method within a child class's constructor is used to call the parent class's constructor and initialize its properties.

Inheritance

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

  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

class Dog extends Animal {        // Child Class
  constructor(name, breed) {
    super(name);                  // Call the parent class constructor
    this.breed = breed;
  }

  bark() {
    console.log(`${this.name} barks!`);
  }
}

const myDog = new Dog("Buddy", "Golden Retriever");
myDog.speak();                                  // Output: Buddy makes a sound. (inherited from Animal)
myDog.bark();                                   // Output: Buddy barks! (specific to Dog)
```

Polymorphism

- Polymorphism in JavaScript, a key concept in object-oriented programming, refers to the ability of different objects to respond to the same method or property name in different ways, depending on their specific type or context.

- Method Overriding: This is the most common form of polymorphism in JavaScript, achieved through its prototype-based inheritance. A subclass (or child class) can provide its own specific implementation of a method that is already defined in its superclass (or parent class). When that method is called on an instance of the subclass, the subclass's implementation is executed, effectively "overriding" the parent's method.

- Polymorphic Functions (Dynamic Typing): JavaScript's dynamic typing allows functions to accept arguments of different types and adapt their behavior based on the actual type of the argument provided at runtime. While not "method overloading" in the strict sense of languages like Java, similar functionality can be achieved by checking argument types or numbers within the function.

Method Overriding

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

// Child class overrides speak()
class Dog extends Animal {
  speak() {
    console.log("The dog barks!");
  }
}

// Using the classes
const genericAnimal = new Animal();
const myDog = new Dog();

genericAnimal.speak();          // Output: The animal makes a sound.
myDog.speak();                  // Output: The dog barks!
```

Polymorphic Functions

```
function displayInfo(item){
    if (typeof item === 'object' && item !== null && 'name' in item){
        console.log(`Name: ${item.name}`);
    } else if (typeof item === 'string'){
        console.log(`Info: ${item}`);
    } else {
        console.log("Unknown item type.");
    }
}

displayInfo({ name: "Alice", age: 30 });        // Outputs: Name: Alice
displayInfo("Hello World");                     // Outputs: Info: Hello World
displayInfo(123);                               // Outputs: Unknown item type.
```

Abstraction

- Abstraction is the OOP concept of hiding complexity by exposing only the relevant parts of an object or system.

- It lets you simplify your code for users of your objects or classes, who don’t need to know the internal details.

- Abstraction is about defining what an object does, not how it does it.

- Unlike languages like Java or C#, JavaScript doesn’t have built-in support for abstract classes or interfaces – but you can still achieve abstraction manually by:

    - Creating methods that provide a simple API for your objects.
    - Keeping internal details hidden using private fields or closures.
    - Using classes to expose only the methods a user should call, hiding implementation details.

Abstraction

```
class B{
    #balance;
    constructor(B1){
        this.#balance = B1;
    }
    deposit(amount){
        this.#balance += amount;
        console.log(`Deposited: $${amount}`);
    }
    getB(){
        return this.#balance;
    }
}

const a1 = new B(1000);
a1.deposit(500);
console.log(a1.getB());
```

THANK YOU!

JS