ARRAY METHODS

Array methods are built-in functions in JavaScript that help you perform various operations on arrays. These methods make it easier to manipulate, search, sort, transform, and work with arrays without having to write repetitive code.

Array.forEach()

The **JavaScript Array forEach() method** is a built-in function that executes a provided function once for each array element. It does not return a new array and does not modify the original array. It's commonly used for iteration and performing actions on each array element.

SYNTAX

```
array.forEach(callback(element, index, arr), thisValue);
```

PARAMETERS

Parameter	Description
callback	It is a callback function executes on each array element.
element	The current element being processed in the array.
index (Optional)	The index of current element. The array indexing starts from 0.

array (Optional)	The array on which forEach() is called.
thisArg (Optional)	Value to use as this when executing the callback function.

EXAMPLE

```
const arr = [1, 2, 3, 4, 5];

arr.forEach((item) => {
    console.log(item);
});
```

OUTPUT

```
1
2
3
4
5
```

Array.map()

The **map() method** is an ES5 feature that creates a new array by applying a function to each element of the original array. It skips empty elements and does not modify the original array.It returns a new array and the arrays' elements result from the callback function.

SYNTAX

```
arr.map((element, index, array) => { /* … */ })
```

PARAMETERS

Parameter	Description
element	t is a required parameter and holds the current element's value.
index	It is an optional parameter and it holds the index of the current element.
arr	t is an optional parameter and it holds the array.

EXAMPLE - Here, we are using the map() method to create a new array containing the square roots of each number in the original array.

```
const a = [1, 4, 9, 16, 25];
const sr = a.map(num => Math.sqrt(a));
```

```
console.log(sr);

});
```

OUTPUT

```
[ NaN, NaN, NaN, NaN, NaN ]
```

EXAMPLE -2 - This example uses the array map() method and returns the square of the array element.

```
let a = [2, 5, 6, 3, 8, 9];


// Using map to transform elements
let res = a.map(function (val, index) {
    return { key: index, value: val * val };
})


console.log(res)
```

OUTPUT

```
[
  { key: 0, value: 4 },
  { key: 1, value: 25 },
  { key: 2, value: 36 },
  { key: 3, value: 9 },
  { key: 4, value: 64 },
  { key: 5, value: 81 }
]
```

Array.filter()

The **filter() method** creates a new array containing elements that satisfy a specified condition. This method skips empty elements and does not change the original array. It returns an array of elements that pass the test and an empty array if no elements pass the test.

SYNTAX

```
array.filter(callback(element, index, arr), thisValue)
```

PARAMETERS

Parameter	Description
callback	The function is to be called for each element of the array.
element	The value of the element currently being processed.
index	(Optional) The index of the current element in the array, starting from 0.
arr	(Optional) The complete array on which Array.every is called.

thisValue	(Optional) The context to be passed as this to be used while executing the callback function. If not provided, undefined is used as the default context.

EXAMPLE - 1 - Creating a new array consisting of only those elements that satisfy the condition checked by **isPositive()** function.

```
function isPositive(value) {
    return value > 0;
}

let filtered = [112, 52, 0, -1, 944].filter(isPositive);
console.log(filtered);
```

OUTPUT

```
[ 112, 52, 944 ]
```

EXAMPLE -2 - Creating a new array consisting of only those elements that satisfy the condition checked by **isEven()** function.

```
function isEven(value) {
    return value % 2 == 0;
}

let filtered = [11, 98, 31, 23, 944].filter(isEven);
```

```
console.log(filtered);
```

OUTPUT

[98, 944]

Array.find()

The find() method in JavaScript looks through an array and returns the first item that meets a specific condition you provide. If no item matches, it returns ***undefined***. It skips any empty space in the array and doesn't alter the original array.

SYNTAX

```
array.find(function(currentValue, index, arr), thisValue)
```

PARAMETERS

Parameter	Description
currentValue	The current element being processed in the array.
index	The index of the current element being processed in the array.
arr	The array find() was called upon.

EXAMPLE - 1 - In this example we searches for the first positive element in the array. The find() method iterates through the array, returning the first element greater than 0. It logs the result to the console.

```
// Input array contain some elements.
let array = [-10, -0.20, 0.30, -40, -50];

// Method (return element > 0).
let found = array.find(function (element) {
    return element > 0;
});

// Printing desired values.
console.log(found);
```

OUTPUT

```
0.3
```

EXAMPLE -2 - In this example we searches for the first element in the array greater than 20. It uses the find() method to iterate through the array and returns the first element that satisfies the condition. Finally, it logs the result (30) to the console.

```
// Input array contain some elements.
let array = [10, 20, 30, 40, 50];


// Method (return element > 10).
let found = array.find(function (element) {
    return element > 20;
});
// Printing desired values.
```

```
console.log(found);
```

OUTPUT

30

Array.reduce()

The **JavaScript Array.reduce() method** iterates over an array, applying a reducer function to each element, accumulating a single output value. It takes an initial value and processes elements from left to right, reducing the array to a single result. It is useful for doing operations like max in an array, min in an array and sum of array

SYNTAX

```
array.reduce( function(total, currentValue, currentIndex, arr), initialValue )
```

PARAMETERS

Parameter	Description
total	Specifies the initial value or previously returned value of the function
currentValue	Specifies the value of the current element
currentIndex	Specifies the array index of the current element
arr	Specifies the array object the current element belongs to

EXAMPLE - 1 - Sum of lengths of all Strings using reduce()

```
const a = ["js", "html", "css"];

// Use reduce to calculate the sum of the lengths of the strings
const res = a.reduce((acc, str) => acc + str.length, 0);

console.log(res);
```

OUTPUT

9

EXAMPLE -2 - sum of the array using reduce

```
const a = [2, 4, 6];


// Use reduce to calculate the sum
const sum = a.reduce((acc, x) => acc + x, 0);


console.log(sum);
```

OUTPUT

```
12
```

PRACTICE QUESTIONS

1. Write a program to find the first number greater than 50 in an array of numbers using the `find` method.
2. Create an array of objects representing students and use the `filter` method to get all students with marks greater than 80.
3. Use the `map` method to create an array of squared values from an array of numbers.
4. Write a program to iterate over an array of names using the `forEach` method and print a greeting for each name.
5. Implement a program to calculate the total price of items in a shopping cart array using the `reduce` method.
6. Use the `filter` method to get all even numbers from an array of integers.
7. Create a program that uses the `map` method to add 5 years to each person's age in an array of objects containing names and ages.
8. Write a program to find the first word longer than 5 characters in an array of strings using the `find` method.
9. Use the `reduce` method to count the occurrences of each word in an array of strings.
10. Create an array of product objects and use `forEach` to print the product name and price in a formatted way.

Call( ) , Apply( ), Bind( ) Methods

This Keyword

The 'this keyword' in JavaScript refers to the object to which it belongs. Its value is determined by how a function is called, making it a dynamic reference. The 'this' keyword is a powerful and fundamental concept used to access properties and methods of an object, allowing for more flexible and reusable code.

EXAMPLE

```
const person = {
    name: "synnefo",
    greet() {
        return `Welcome To, ${this.name}`;
    }
```

```
};
console.log(person.greet());


OUTPUT

Welcome To, synnefo
```

Using this in a method

EXAMPLE - In the context of an object method in JavaScript, the this keyword refers to the object itself, allowing access to its properties and methods within the method's scope. It facilitates interaction with the object's data and behaviour, providing a way to access and manipulate its state.

```
const person = {
    name: 'John',
    age: 30,
    greet() {
        console.log('Hello, my name is ' +
            this.name + ' and I am '
            + this.age +
            ' years old.');
    }
};

person.greet();


OUTPUT
Hello, my name is John and I am 30 years old.
```

Using this in a function

In a JavaScript function, the behavior of the this keyword varies depending on how the function is invoked.

EXAMPLE

```
function greet() {
    console.log('Hello, my name is ' + this.name);
```

```
}

const person = {
    name: 'Amit',
    sayHello: greet
};
const anotherPerson = {
    name: 'Jatin'
};


//Driver Code Starts{
greet();
person.sayHello();
greet.call(anotherPerson);
//Driver Code Ends }



OUTPUT
Hello, my name is undefined
Hello, my name is Amit
Hello, my name is Jatin
```

Using this alone

When used alone in JavaScript, outside of any specific context, the behavior of the this keyword depends on whether the code is running in strict mode or not.

```
console.log(this);

OUTPUT
{}
```

Call() Method

The **Call() Method** calls the function directly and sets **this** to the first argument passed to the call method and if any other sequences of arguments preceding the first argument are passed to the call method then they are passed as an argument to the function.The call method doesn't return a new function.

This is exactly the same as the MyBind function but this only doesn't return a function because the call method given by javascript also doesn't return a method. So while implementing the polyfill of call we also need to keep this in mind.

SYNTAX

```
call(objectInstance)
call(objectInstance, arg1, /* …, */ argN)
```

EXAMPLE -1 - Before implementing our own call polyfill let us see the call method which is given by javascript

```
let nameObj = {
    name: "Tony"
}

let PrintName = {
    name: "steve",
    sayHi: function (age) {
        console.log(this.name + " age is " + age);
    }
}

PrintName.sayHi.call(nameObj, 42);
```

OUTPUT

```
Tony age is 42
```

EXAMPLE -2 - Now let us write our own call polyfill

```
let nameObj = {
    name: "Tony"
}

let PrintName = {
    name: "steve",
    sayHi: function () {
        console.log(this.name);
    }
}
```

```
Object.prototype.MyBind = function (bindObj) {

    // Here "this" will be sayHi function
    bindObj.myMethod = this;
    return function () {
        bindObj.myMethod();
    }
}
let HiFun = PrintName.sayHi.MyBind(nameObj);
HiFun();
```

OUTPUT

Tony

Apply() Method

The **Apply() Method** calls the function directly and sets **this** to the first argument passed to the apply method and if any other arguments provided as an array are passed to the call method then they are passed as an argument to the function.

SYNTAX

```
apply(objectInstance)
apply(objectInstance, argsArray)
```

EXAMPLE -1 - For the final time let us see apply method given by javascript:

```
let nameObj = {
    name: "Tony"
}
```

```
let PrintName = {
    name: "steve",
    sayHi: function (...age) {
        console.log(this.name + " age is " + age);
    }
}
PrintName.sayHi.apply(nameObj, [42]);
```

OUTPUT

Tony age is 42

EXAMPLE -2 - Now let us write our final polyfill which is apply polyfill:

```
let nameObj = {
    name: "Tony"
}

let PrintName = {
    name: "steve",
    sayHi: function (age) {
        console.log(this.name + " age is " + age);
    }
}

Object.prototype.MyApply = function (bindObj, args) {
    bindObj.myMethod = this;

    bindObj.myMethod(...args);

}
PrintName.sayHi.MyApply(nameObj, [42]);
```

OUTPUT

Tony age is 42

Bind() Method

The **Bind() Method** creates a new function and when that new function is called it set **this** keyword to the first argument which is passed to the bind method, and if any other sequences of arguments preceding the first argument are passed to the bind method then they are passed as an argument to the new function when the new function is called.

SYNTAX

```
bind(thisArg)
bind(thisArg, arg1, arg2, /* …, */ argN)
```

EXAMPLE -1 - Let us first see what will be the actual output with the bind method that is given by javascript:

```
let nameObj = {
    name: "Tony"
}
```

```
let PrintName = {
    name: "steve",
    sayHi: function () {

        // Here "this" points to nameObj
        console.log(this.name);
    }
}

let HiFun = PrintName.sayHi.bind(nameObj);
HiFun();
```

OUTPUT

Tony

EXAMPLE -2 - Now let us write our own bind polyfill. We will implement our bind polyfill using a prototype on the Object class in the above example:

```
let nameObj = {
    name: "Tony"
}

let PrintName = {
    name: "steve",
    sayHi: function () {
        console.log(this.name);
    }
}

Object.prototype.MyBind = function (bindObj) {

    // Here "this" will be sayHi function
    bindObj.myMethod = this;
    return function () {
        bindObj.myMethod();
    }
}
let HiFun = PrintName.sayHi.MyBind(nameObj);
```

```
HiFun();
```

OUTPUT

Tony

PRACTICE QUESTIONS

1. Write a function that calculates the area of a rectangle and use call to pass different objects with length and width properties.
2. Create a method in one object and use apply to invoke it on another object, passing an array of arguments.
3. Use bind to create a new function that always logs a specific user's name when called.
4. Implement a function that calculates the sum of multiple numbers and use apply to pass an array of numbers dynamically.
5. Create an object with a greet method and use call to greet users with names from a different object.
6. Use bind to attach a click event listener to a button, ensuring the callback function always refers to the correct object context.
7. Write a function that updates an object's properties and use call to apply the function to multiple different objects.
8. Implement a function with default parameters and use bind to create a partially applied version of the function.
9. Create a reusable function that formats a date string and use apply to invoke it with different date arguments.
10. Write a function that logs a custom message with a prefix and use bind to predefine the prefix for later usage.

OBJECT ORIENTED PROGRAMMING IN JAVASCRIPT

CLASSES

Classes in JavaScript are a blueprint for creating objects, introduced in ES6. They encapsulate data and behavior by defining properties and methods, enabling object-oriented programming. Classes simplify the creation of objects and inheritance, making code more organized and reusable.

SYNTAX

```
class ClassName {
 // Constructor method
constructor(parameters) {
// Initialization code,
setting up properties
this.property = value; } // Method definitions
methodName()
```

```
{ // Method logic }
}
```

EXAMPLE -1 -The Emp class initializes name and age properties for each new instance using a constructor.

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

const emp = new Emp("Aman", "25 years");
console.log(emp.name);
console.log(emp.age);
```

OUTPUT

```
Aman
25 years
```

CONSTRUCTOR

The constructor method is a special method used for initializing objects created with a class. It’s called automatically when a new instance of the class is created. It typically assigns initial values to object properties using parameters passed to it. This ensures objects are properly initialized upon creation.

EXAMPLE -

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

```
const p1 = new Person("Alice", 30);
console.log(p1.name);
console.log(p1.age);

const p2 = new Person("Bob", 25);
console.log(p2.name);
console.log(p2.age);

OUTPUT

Alice
30
Bob
25
```

OBJECT

An Object is a unique entity that contains properties and methods. For example “a car” is a real-life Object, which has some characteristics like color, type, model, and horsepower and performs certain actions like driving. The characteristics of an Object are called Properties in Object-Oriented Programming and the actions are called methods. An Object is an instance of a class. Objects are everywhere in JavaScript, almost every element is an Object whether it is a function, array, or string.

The object can be created in two ways in JavaScript:

- Object Literal
- Object Constructor

EXAMPLE - using object Literal

```
// Defining object

let person = {
```

```
        first_name: 'Mukul',
        last_name: 'Latiyan',


        //method
        getFunction: function () {
                return (`The name of the person is
                ${person.first_name} ${person.last_name}`)
        },
        //object within object
        phone_number: {
                mobile: '12345',
                landline: '6789'
        }
}
console.log(person.getFunction());
console.log(person.phone_number.landline);
```

EXAMPLE - using object Constructor

```
// Using a constructor
function person(first_name, last_name) {
```

```
        this.first_name = first_name;

        this.last_name = last_name;

}

// Creating new instances of person object

let person1 = new person('Mukul', 'Latiyan');

let person2 = new person('Rahul', 'Avasthi');



console.log(person1.first_name);

console.log(`${person2.first_name} ${person2.last_name}`);
```

Note: The JavaScript Object.create() Method creates a new object, using an existing object as the prototype of the newly created object.

EXAMPLE -

```
// Object.create() example a

// simple object with some properties

const coder = {

        isStudying: false,
```

```
        printIntroduction: function () {

                console.log(`My name is ${this.name}. Am I

                        studying?: ${this.isStudying}.`)

        }

}

// Object.create() method

const me = Object.create(coder);

// "name" is a property set on "me", but not on "coder"

me.name = 'Mukul';

// Inherited properties can be overwritten

me.isStudying = true;



me.printIntroduction();
```

GETTERS AND SETTERS

One of the many features of JavaScript is its ability to define getters and setters for object properties. Getters and setters provide a way to encapsulate the implementation details of an object property, while still allowing external code to

access and modify its value. In this article, we'll explore how to define and use getters and setters in JavaScript.

Getters and setters are defined using the get and set keywords respectively. When you define a getter or setter, you are actually defining a method that is associated with a particular property of an object. Here's an example of how to define a getter and setter for a person object:

EXAMPLE -

```
const person = {
	firstName: "John",
	lastName: "Doe",
	get fullName() {
		return `${this.firstName} ${this.lastName}`;
	},
	set fullName(name) {
		const parts = name.split(" ");
		this.firstName = parts[0];
		this.lastName = parts[1];
	},
};

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

person.fullName = "Jane Smith";
console.log(person.firstName); // "Jane"
console.log(person.lastName); // "Smith"
```

OUTPUT

```
John Doe
Jane
Smith
```

EXAMPLE -2 - Temperature Conversion: Let's say we have a temperature object that stores the temperature value in Celsius. We can define a getter and setter for this object that allows us to get and set the temperature value in Fahrenheit. Here's how we can do this:

```
const temperature = {
        _celsius: 0,
        get fahrenheit() {
                return this._celsius * 1.8 + 32;
        },
        set fahrenheit(value) {
                this._celsius = (value - 32) / 1.8;
        },
};

console.log(temperature.fahrenheit); // 32
temperature.fahrenheit = 68;
console.log(temperature._celsius); // 20
```

OUTPUT

```
32
20
```

INHERITANCE

JavaScript inheritance is the method through which the objects inherit the properties and the methods from the other objects. It enables code reuse and structuring of relationships between objects, creating a hierarchy where a child object can access features of its parent object

PROTOTYPAL INHERITANCE

Objects inherit from other objects through their prototypes. Each object has a prototype, properties, and methods inherited from that prototype. The methods through which prototypal inheritance

EXAMPLE (PROTOTYPAL INHERITANCE)

```
function Animal(name) {
```

```
    this.name = name;
}

Animal.prototype.sound = function() {
    console.log(";Some generic sound”);
};

function Dog(name, breed) {
    Animal.call(this, name);
    this.breed = breed;
}

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

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

const myDog = new Dog("Buddy", "Labrador");
myDog.sound();  // Outputs: Woof! Woof!
```
OUTPUT
Woof! Woof!

CLASSICAL INHERITANCE

Introduced in ECMAScript6 (ES6) with the class keyword. Uses a class-based approach similar to other programming languages like Java or C++. Following are the methods through which class-based inheritance is achieved in JavaScript

JavaScript ES6 classes support the extended keyword to perform class inheritance.

EXAMPLE - Demonstrating class inheritance and method overloading in JavaScript.

```
class automobile {
    constructor(name, cc) {
        this.name = name;
        this.cc = cc;
    }
    engine() {
        console.log(`${this.name}
```

```
      has ${this.cc} engine`);
    }
}

class car extends automobile {
    engine() {
        console.log(this.name,
            ";has ";, this.cc, ";cc engine";);
    }
}

let carz = new car('Rex', ";1149";);
carz.engine();
```

OUTPUT

```
Rex has  1149 cc engine
```

INHERITANCE USING THE SUPER KEYWORD

Super keyword is used in classes to call the properties and methods of the parent class

EXAMPLE -Using super keyword for method invocation and inheritance in JavaScript.

```
/ Inheritance using super keyword in JS
class Automobile {
    constructor(name) {
        this.name = name;
    }

    engine() {
        console.log(this.name,
            ";has ";, this.cc, ";cc engine";);
    }
}

class Car extends Automobile {
    constructor(name, cc) {
        super(name);

        // Additional properties for
```

```
        // the Car class
        this.cc = cc;
    }

    engine() {
        // the 'engine' method of the parent
        // class using 'super'
        super.engine();

        console.log(this.name,
            ';has ';, this.cc, ';cc engine';);
    }
}

let carz = new Car('Rexton', '1500');
carz.engine();
```

OUTPUT

```
Rexton has  1500 cc engine
Rexton has  1500 cc engine
```

FUNCTIONAL INHERITANCE

Objects inherit properties and methods from other objects through function constructors. It uses functions to create objects and establish relationships between them. The methods through which functional inheritance is achieved in JavaScript are as follows:

Constructor overriding

In JavaScript, when we want to extend a class we might want to override the constructor using the super keyword which invokes the parent constructor.

EXAMPLE -

```
function Animal(name) {
    const obj = {};
```

```
    obj.name = name;

    obj.sound = function() {
        console.log(";Some generic sound";);
    };

    return obj;
}

function Dog(name, breed) {
    const obj = Animal(name);
    obj.breed = breed;

    obj.sound = function() {
        console.log(";Woof! Woof!";);
    };

    return obj;
}

const myDog = Dog(";Buddy";, ";Labrador";);
myDog.sound();
```

OUTPUT

```
Woof! Woof!
```

ABSTRACTION

JavaScript abstraction refers to the concept of hiding complex implementation details and showing only the essential features or functionalities of an object or module to the user also it is the fundamental concept in object-oriented programming.

Creating abstraction in JavaScript involves organizing your code in a way that hides complex details and exposes only the essential features to other parts of your program.

Achieving abstraction in JavaScript involves creating **abstract classes and interfaces**, even though JavaScript itself doesn't have native support for these

concepts. Instead, developers often use prototypes, functions, and object-oriented patterns to enforce abstraction.

EXAMPLE - Here, the Animal abstract class has an abstract method makeSound, and the Dog class extends Animal, providing a concrete implementation for the makeSound method. Trying to instantiate an object of the abstract class Animal will throw an error, showing the abstraction concept.

```
Function Animal() {
    if (this.constructor === Animal) {
        throw new Error(`Cannot instantiate
        abstract class Animal`);
    }

    this.makeSound = function () {
        throw new Error(`Cannot call abstract
        method makeSound from Animal`);
    };
}

// Create a concrete class Dog that extends Animal
function Dog(name) {
    Animal.call(this);
    this.name = name;

    this.makeSound = function () {
        console.log(`${this.name} barks`);
    };
}

// Inherit from the abstract class
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

// Create an instance of the Dog class
let dog = new Dog(&amp;quot;Buddy&amp;quot;);
dog.makeSound();

// Try to create an instance of
// the abstract class Animal
try {
    let animal = new Animal();
```

```
} catch (error) {
    console.error(error.message);
}

OUTPUT
Cannot instantiate abstract class Animal
```

EXAMPLE - Here, the Shape class serves as an abstract class with a property shapeName and a method display. The Triangle class extends Shape and provides a concrete implementation for the shapeName property. Creating an object of the abstract class Shape will result in an error, enforcing the abstraction concept.

```
// Creating a constructor function
// for the abstract class Shape
function Shape() {
    this.shapeName = &amp;quot;shapeName&amp;quot;;
    throw new Error(`You cannot create an
    instance of Abstract Class`);
}

Shape.prototype.display = function () {
    return &amp;quot;Shape is: &amp;quot; + this.shapeName;
};

// Creating a constructor function
// for the concrete class Triangle
function Triangle(shapeName) {
    this.shapeName = shapeName;
}

// Creating an object without
// using the function constructor
Triangle.prototype = Object
    .create(Shape.prototype);

// Creating an instance of the Triangle class
let triangle = new Triangle(&amp;quot;Equilateral&amp;quot;);
console.log(triangle.display());

OUTPUT

Cannot instantiate abstract class Animal
```

ENCAPSULATION

Encapsulation is a fundamental concept in object-oriented programming that refers to the practice of hiding the internal details of an object and exposing only the necessary information to the outside world.

Using Closures

In JavaScript, closures are functions that have access to variables in their outer lexical environment, even after the outer function has returned. Private variables and methods can be created using closures.

EXAMPLE -In this example, we have created a BankAccount object using a closure. The object has three private variables: _accountNumber, _accountHolderName, and _balance. These variables are only accessible within the BankAccount function and cannot be accessed from outside. The showAccountDetails function is a private method that displays the account details. The deposit and withdrawal methods are public methods that can be accessed from outside the object. When these methods are called, they update the _balance variable and call the showAccountDetails function to display the updated account details.

```
function BankAccount(accountNumber, accountHolderName, balance) {
    let _accountNumber = accountNumber;
    let _accountHolderName = accountHolderName;
    let _balance = balance;

    function showAccountDetails() {
        console.log(`Account Number: ${_accountNumber}`);
        console.log(`Account Holder Name: ${_accountHolderName}`);
        console.log(`Balance: ${_balance}`);
    }
```

```
    function deposit(amount) {
        _balance += amount;
        showAccountDetails();
    }

    function withdraw(amount) {
        if (_balance >= amount) {
            _balance -= amount;
            showAccountDetails();
        } else {
            console.log("Insufficient Balance");
        }
    }

    return {
        deposit: deposit,
        withdraw: withdraw
    };
}

let myBankAccount = BankAccount("123456", "John Doe", 1000);

myBankAccount.deposit(500);
// Output: Account Number: 123456 Account Holder Name:
//John Doe Balance: 1500

myBankAccount.withdraw(2000); // Output: Insufficient Balance
```

OUTPUT

```
Account Number: 123456
Account Holder Name: John Doe
Balance: 1500
Insufficient Balance
```

Using Classes

ES6 introduced the class syntax in JavaScript, which allows us to define classes and objects in a more structured way. Classes can be used to achieve encapsulation in JavaScript.

EXAMPLE -In this example, we have created a BankAccount class using the class keyword. The class has three private variables: _accountNumber, _accountHolderName, and _balance. These variables are prefixed with an underscore to indicate that they are private variables. The showAccountDetails method is a public method that displays the account details. The deposit and withdrawal methods are also public methods that can be accessed from outside the object. When these methods are called, they update the _balance variable and call the showAccountDetails method to display the updated account details.

```
class BankAccount {
    constructor(accountNumber, accountHolderName, balance) {
        this._accountNumber = accountNumber;
        this._accountHolderName = accountHolderName;
        this._balance = balance;
    }

    showAccountDetails() {
        console.log(`Account Number: ${this._accountNumber}`);
        console.log(`Account Holder Name: ${this._accountHolderName}`);
        console.log(`Balance: ${this._balance}`);
    }

    deposit(amount) {
        this._balance += amount;
        this.showAccountDetails();
    }

    withdraw(amount) {
        if (this._balance >= amount) {
            this._balance -= amount;
            this.showAccountDetails();
        } else {
            console.log("Insufficient Balance");
        }
    }
}
let myBankAccount = new BankAccount("123456", "John Doe", 1000);
myBankAccount.deposit(500);
// Output: Account Number: 123456 Account Holder Name:
```

```
//John Doe Balance: 150
```

OUTPUT

```
Account Number: 123456
Account Holder Name: John Doe
Balance: 1500
```

POLYMORPHISM

Polymorphism is one of the core concepts of object-oriented programming languages where **poly** means **many** and **morphism** means **transforming one form into another**. Polymorphism means the same function with different signatures is called many times.

In JavaScript, polymorphism works in two primary ways:

- **Method Overriding:** A child class overrides a method of its parent class.
- **Method Overloading (simulated):** A function behaves differently based on the number or type of its arguments.

METHOD OVERRIDING

EXAMPLE - In method overriding, a subclass provides its own implementation for a method defined in the superclass.

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

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

```
class Cat extends Animal {
  speak() {
    console.log("The cat meows.");
  }
}

const animals = [new Animal(), new Dog(), new Cat()];

animals.forEach(animal => animal.speak());
```

OUTPUT

```
This animal makes a sound.
The dog barks.
The cat meows.
```

METHOD OVERLOADING

JavaScript does not natively support method overloading, but you can achieve similar functionality using default parameters, conditional logic, or the arguments object.

EXAMPLE -

```
class Calculator {
      add(a, b, c = 0) {
        return a + b + c;
      }
}
const calc = new Calculator();
console.log(calc.add(2, 3));
console.log(calc.add(2, 3, 4));

OUTPUT
5
9
```

PRACTICE QUESTIONS

1. Create a class for a library system where you can add books and track their availability status.

2. Design an object-oriented solution for a car dealership that includes different types of vehicles with shared and specific properties using inheritance.
3. Implement a banking application with classes for Account, SavingsAccount, and CurrentAccount, demonstrating encapsulation to protect sensitive data.
4. Build a system for a school that includes classes for Student, Teacher, and Administrator, showcasing polymorphism in their respective roles.
5. Create a prototype-based implementation of a game character with methods for movement and attack, then extend it to add specific abilities for different character types.
6. Write a program to simulate a zoo with a base class Animal and derived classes for specific animals, using abstraction to define shared behaviors.
7. Develop a shopping cart system with objects for products, discounts, and a cart, focusing on encapsulation to restrict direct access to internal properties.
8. Create a prototype for a music player with basic methods like play, pause, and stop, and extend it to include playlist management.
9. Implement a ticket booking system for an event with classes for User, Event, and Ticket, using inheritance to handle different types of users (e.g., VIP, regular).
10. Design a customer feedback system with a base class Feedback and subclasses for different feedback types (e.g., complaint, suggestion), demonstrating polymorphism in handling responses.