1. Introduction to JavaScript

a). What is JavaScript?

Definition:
JavaScript is a powerful, lightweight, and interpreted programming language used to make web pages interactive. It allows developers to create dynamic content, handle events, and perform various operations on web pages.

Syntax Example:

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

Examples:

1. **Displaying a message in an alert box:**

   ```
   alert("Welcome to JavaScript!");
   ```

2. **Changing web page content dynamically:**

   ```
   document.getElementById("demo").innerHTML = "Hello, World!";
   ```

b). History of JavaScript

JavaScript was created by Brendan Eich in 1995 while he was working at Netscape Communications. Initially, it was called Mocha, then renamed to LiveScript, and finally to JavaScript to capitalize on Java's popularity. Over the years, JavaScript has evolved significantly with the introduction of ES6 (ECMAScript 2015) and subsequent versions, bringing modern features and improvements.

Key milestones in JavaScript's history:

- **1995:** JavaScript created by Brendan Eich at Netscape.
- **1996:** Microsoft introduced JScript, an alternative version of JavaScript.
- **1997:** ECMAScript standard was established.
- **2009:** Node.js was introduced, enabling server-side JavaScript.
- **2015:** ES6 (ECMAScript 2015) introduced major improvements like `let`, `const`, arrow functions, and promises.

Example of modern ES6 syntax:

```
const greet = (name) => `Hello, ${name}!`;
console.log(greet("Alice"));
```

C). Purpose of JavaScript

JavaScript is used for:

- **Creating Interactive Web Pages:** Adding animations, form validations, and dynamic elements.
- **Handling Events:** Performing actions based on user interactions like clicks or key presses.
- **Asynchronous Operations:** Fetching data from servers without reloading pages.
- **Server-Side Development:** Using frameworks like Node.js to build backend applications.

Syntax Example:

```
function greet(name) {
    return "Hello, " + name;
}
console.log(greet("Alice"));
```

Examples:

1. Validating user input:

```
function validateAge(age) {
    if (age >= 18) {
        return "Eligible";
    } else {
        return "Not Eligible";
    }
}
console.log(validateAge(20));
```

2. Updating page content dynamically:

```
document.querySelector("#title").textContent = "Updated Title";
```

d). Features of JavaScript

1. **Lightweight:** Minimal syntax and runs directly in the browser.
2. **Cross-Platform:** Works across different operating systems and browsers.
3. **Event-Driven:** Handles user interactions effectively.
4. **Versatile:** Can be used for both frontend and backend development.
5. **Supports Asynchronous Programming:** Allows non-blocking execution with promises and async/await.

Syntax Example:

```
let user = "John";
console.log(`Welcome, ${user}!`);
```

Examples:

1. Handling button click event:

```
document.getElementById("btn").addEventListener("click", function() {
    alert("Button Clicked!");
});
```

2. Looping through an array:

```
let colors = ["Red", "Green", "Blue"];
colors.forEach(color => console.log(color));
```

e). JavaScript Program Execution

JavaScript code can be executed in different environments:

1. **Web Browsers:** Run JS inside `<script>` tags in HTML files.
2. **Node.js:** Run JS outside browsers for backend development.
3. **Browser Console:** Directly run JS code for testing and debugging.

Syntax Example (Browser):

```
<script>
    document.write("Hello from JavaScript!");
</script>
```

Examples:

1. In an HTML file:

```
<script>
    console.log("JavaScript inside HTML!");
</script>
```

2. Using Node.js:

```
console.log("Running JavaScript in Node.js");
```

2. Installation of Node.js

a). What is Node.js?

Node.js is a runtime environment that allows JavaScript to be executed outside the browser. It enables server-side scripting and builds scalable applications.

Syntax Example (Node.js script):

```
console.log("Welcome to Node.js");
```

b). Steps to Install Node.js:

1. Visit the official website: https://nodejs.org.
2. Download the appropriate version for your OS (LTS recommended).
3. Install the downloaded file by following on-screen instructions.
4. Verify the installation by running the command:
   node -v

Examples:

1. Running a simple script:

```
console.log("Node.js is installed successfully!");
```

2. Creating a basic server:

```
const http = require('http');
http.createServer((req, res) => {
    res.write("Hello, Node.js!");
    res.end();
}).listen(3000);
console.log("Server running at http://localhost:3000");
```

3.Variables

a). Declaration

Declaring a variable means reserving memory space for storing data.

Syntax:

```
var x;
let y;
const z = 10;
```

b). Assignment

Assigning a value to a previously declared variable.

Syntax:

```
x = 5;
y = "Hello";
```

c). Initialization

Declaring and assigning a value at the same time.

Syntax:

```
let name = "Alice";
const age = 25;
```

d). Scope Statement

Scope determines the accessibility of variables:

- **Global Scope:** Accessible throughout the code.
- **Function Scope:** Accessible within the function.
- **Block Scope:** Accessible within `{}` blocks.

e). Var, Let, and Const

1. `var` (Function-scoped, can be redeclared)

Definition: `var` is function-scoped, meaning it is accessible within the function where it is declared. It can be redeclared and updated within the same scope.

Examples:

```
var a = 10;
var a = 20; // Redeclaration is allowed
console.log(a); // Output: 20
function example() {
    if (true) {
        var test = "Hello";
    }
    console.log(test); // Output: Hello (accessible outside the block)
}
example();
```

2. `let` (Block-scoped, cannot be redeclared)

Definition: `let` is block-scoped, meaning it is accessible only within the block where it is declared. It cannot be redeclared within the same scope but can be updated.

Examples:

```
let b = 10;
b = 20; // Allowed (reassignment)
console.log(b); // Output: 20
if (true) {
    let name = "Alice";
    console.log(name); // Output: Alice
}
// console.log(name); // Error: name is not defined outside block
```

3. `const` (Block-scoped, cannot be changed)

Definition: `const` is block-scoped and used to declare variables with constant values that cannot be reassigned after initialization.

Examples:

```
const c = 10;
// c = 20; // Error: Assignment to constant variable
console.log(c); // Output: 10
const person = { name: "John" };
person.name = "Doe"; // Allowed (modifying object properties)
console.log(person.name); // Output: Doe
```

4. Data Types

JavaScript provides several data types:

a). Number

Represents both integers and floating-point numbers.

Example:

```
let num = 42;
let price = 99.99;
```

b). Boolean

Represents true or false values.

Example:

```
let isAvailable = true;
```

c). String

Represents sequences of characters.

Example:

```
let message = "Hello, World!";
```

d). Null

Represents an empty or non-existent value.

Example:

```
let emptyValue = null;
```

e). Undefined

Indicates a variable has been declared but not assigned a value.

Example:

```
let notAssigned;
```

f). BigInt

Used for large integer values.

Example:

```
let bigNumber = 1234567890123456789012345678890n;
```

g). Symbol

Represents a unique identifier.

Example:

```
let uniqueKey = Symbol('id');
```

8. Object

Used to store collections of data in key-value pairs.

Example:

```
let person = {
    name: "John",
    age: 30
};
```

5. Functions

a). What is a Function?

A function is a reusable block of code designed to perform a particular task. It helps in modularizing the code and enhancing reusability.

Example:

```
function greet() {
    console.log("Hello, World!");
}
greet();
```

b). Purpose of Functions

- Code reusability
- Modular structure
- Improved readability
- Easier debugging and maintenance

c). Syntax to Create a Function

```
function functionName(parameters) {
    // Function body
    return result;
}
```

d). Function Definition

A function is defined using the `function` keyword, followed by the function name and parentheses.

e). Function Scope

Functions can have local scope (variables declared inside) or global scope (accessible everywhere).

f). Function Block

The function block is enclosed within curly braces `{}` containing the code to be executed.

g). Function Call

A function is executed when it is called by its name followed by parentheses.

Example:

```
function sayHello() {
    console.log("Hello!");
}
sayHello();
```

h). Types of Functions

1. Normal Function

A standard function declaration using the `function` keyword.

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

2. Parameters and Arguments

Parameters are placeholders used in function definition, while arguments are actual values passed when calling the function.

```
function multiply(x, y) {
    return x * y;
}
console.log(multiply(4, 2));
```

3. Return Statement

Functions can return a value using the `return` keyword.

```
function square(num) {
    return num * num;
}
console.log(square(6));
```

4. Callback Function

A function passed as an argument to another function.

```
function process(callback) {
    callback();
}
process(() => console.log("Callback executed"));
```

5. Function Expression

A function assigned to a variable.

```
const greet = function() {
    console.log("Hello there!");
};
greet();
```

6. Anonymous Function

A function without a name, often used as an argument.

```
setTimeout(function() {
    console.log("This is an anonymous function");
}, 1000);
```

7. Arrow Function

```
A concise way to write functions using => syntax.
const sum = (a, b) => a + b;
console.log(sum(3, 5));
```

8. Async Function

Used to handle asynchronous operations.

```
async function fetchData() {
    return "Data received";
}
fetchData().then(console.log);
```

9. Higher-Order Function

A function that takes another function as an argument or returns a function.

```
function operate(operation, x, y) {
    return operation(x, y);
}
console.log(operate((a, b) => a + b, 5, 10));
```

6. Object

a). What is an Object?

An object is a collection of key-value pairs that represent data and behavior. Objects allow you to store and manage related data efficiently.

Example:

```
const person = {
    name: "Alice",
    age: 25,
    greet: function() {
        console.log("Hello, " + this.name);
    }
};
person.greet();
```

b.) Purpose of Objects

- Organize data into a structured format
- Encapsulate related data and functionality

- Improve code reusability and readability

c). Creation of Objects

Objects can be created using object literals or the `new Object()` constructor.

Example 1 (Object Literal):

```
const car = {
    brand: "Toyota",
    model: "Corolla",
    year: 2022
};
```

Example 2 (Constructor):

```
const car = new Object();
car.brand = "Toyota";
car.model = "Corolla";
car.year = 2022;
```

d). Properties

Properties are the key-value pairs stored inside an object.

Example:

```
console.log(car.brand);  // Output: Toyota
```

e). CRUD Operations on Objects

i). Read

Access object properties using dot notation or bracket notation.

```
console.log(person.name); // Dot notation
console.log(person["age"]); // Bracket notation
```

ii). Insert

Add new properties to an object.

```
person.city = "New York";
console.log(person);
```

iii). Update

Modify existing properties.

```
person.age = 30;
console.log(person);
```

iv). Delete

Remove a property from an object.

```
delete person.city;
console.log(person);
```

f). Object Methods

i). Object.seal()

Prevents new properties from being added to an object but allows modification of existing properties.

```
const user = { name: "John" };
Object.seal(user);
user.name = "Doe"; // Allowed
user.age = 30;  // Not allowed
console.log(user);
```

ii). Object.assign()

Copies properties from one or more source objects to a target object.

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

iii). Object.keys()

Returns an array of an object's property names.

```
const keys = Object.keys(person);
console.log(keys);
```

7. Array

a). Syntax

An array in JavaScript is a special type of object used for storing ordered collections of values.

Definition: Arrays are lists or ordered collections of items.

Syntax:

```
let array = [item1, item2, item3, ...];
```

b). Purpose

Arrays allow you to store multiple values in a single variable and are used when you need to store a list of items like numbers, strings, objects, etc.

c). Literal Notation

Literal notation is the most common way to create arrays.

Syntax:

```
let array = [1, 2, 3, 4];
```

d). Index

Arrays are zero-indexed, meaning the first element is at index 0.

Example 1:

```
let fruits = ['Apple', 'Banana', 'Orange'];
console.log(fruits[0]); // Output: Apple
```

Example 2:

```
let numbers = [10, 20, 30];
console.log(numbers[2]); // Output: 30
```

8. Array Methods

a). Push

Adds one or more elements to the end of an array and returns the new length of the array.

Syntax:

```
array.push(element1, element2, ...);
```

Example 1:

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

Example 2:

```
et colors = ['red', 'blue'];
colors.push('green', 'yellow');
console.log(colors); // Output: ['red', 'blue', 'green', 'yellow']
```

b). Pop

Removes the last element from an array and returns that element.

Syntax:

```
array.pop();
```

Example 1:

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

Example 2:

```
let names = ['Alice', 'Bob', 'Charlie'];
let poppedName = names.pop();
console.log(poppedName); // Output: Charlie
console.log(names);      // Output: ['Alice', 'Bob']
```

c). Shift

Removes the first element from an array and returns that element.

Syntax:

```
array.shift();
```

Example 1:

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

Example 2:

```
let animals = ['dog', 'cat', 'rabbit'];
let removedAnimal = animals.shift();
console.log(removedAnimal); // Output: dog
console.log(animals);      // Output: ['cat', 'rabbit']
```

d). Unshift

Adds one or more elements to the beginning of an array and returns the new length of the array.

Syntax:

```
array.unshift(element1, element2, ...);
```

Example 1:

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

Example 2:

```
let letters = ['b', 'c'];
letters.unshift('a');
console.log(letters); // Output: ['a', 'b', 'c']
```

e). ForEach

Executes a provided function once for each array element.

Syntax:

```
array.forEach(callback(currentValue, index, array));
```

Example 1:

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

Example 2:

```
let names = ['John', 'Jane', 'Doe'];
names.forEach(name => console.log(name));
// Output: John, Jane, Doe
```

f). Map

Creates a new array populated with the results of calling a provided function on every element in the array.

Syntax:

```
let newArray = array.map(callback(currentValue, index, array));
```

Example 1:

```
let arr = [1, 2, 3];
let squared = arr.map(x => x * x);
console.log(squared); // Output: [1, 4, 9]
```

Example 2:

```
let names = ['John', 'Jane'];
let upperNames = names.map(name => name.toUpperCase());
console.log(upperNames); // Output: ['JOHN', 'JANE']
```

h). Filter

Creates a new array with all elements that pass the test implemented by the provided function.

Syntax:

```
let newArray = array.filter(callback(currentValue, index, array));
```

Example 1:

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

Example 2:

```
let numbers = [10, 15, 20, 25];
let numbersGreaterThan15 = numbers.filter(num => num > 15);
console.log(numbersGreaterThan15); // Output: [20, 25]
```

i). Splice

Changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.

Syntax:

```
array.splice(start, deleteCount, item1, item2, ...);
```

Example 1:

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

Example 2:

```
let animals = ['cat', 'dog', 'rabbit'];
animals.splice(1, 1, 'hamster');
console.log(animals); // Output: ['cat', 'hamster', 'rabbit']
```

j). Slice

Returns a shallow copy of a portion of an array into a new array object selected from the start to the end (end not included).

Syntax:

```
let newArray = array.slice(start, end);
```

Example 1:

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

Example 2:

```
let colors = ['red', 'green', 'blue', 'yellow'];
let colorSlice = colors.slice(2);
console.log(colorSlice); // Output: ['blue', 'yellow']
```

k). Includes

Determines whether an array contains a certain element.

Syntax:

```
array.includes(element, start);
```

Example 1:

```
let arr = [1, 2, 3];
console.log(arr.includes(2)); // Output: true
```

Example 2:

```
let names = ['Alice', 'Bob', 'Charlie'];
console.log(names.includes('Bob')); // Output: true
```

I). indexOf

Returns the first index at which a given element can be found in the array, or -1 if it is not present.

Syntax:

```
array.indexOf(element, start);
```

Example 1:

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

Example 2:

```
let names = ['John', 'Jane', 'Doe'];
console.log(names.indexOf('Jane')); // Output: 1
```

9. Selection Statements

a). If

The `if` statement evaluates a condition, and if the condition is true, it executes the code block inside it.

Syntax:

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

Example 1:

```
let age = 18;
if (age >= 18) {
  console.log('You are an adult.');
}
// Output: You are an adult.
```

Example 2

```
let temperature = 30;
if (temperature > 25) {
  console.log('It\'s a hot day!');
}
// Output: It's a hot day!
```

b). Else

The `else` statement provides an alternative code block to be executed if the condition in the `if` statement is false.

Syntax:

```
if (condition) {
  // code if condition is true
} else {
```

```
  // code if condition is false
}
```

Example 1:

```
let age = 16;
if (age >= 18) {
  console.log('You are an adult.');
} else {
  console.log('You are a minor.');
}
// Output: You are a minor.
```

Example 2:

```
let isRaining = false;
if (isRaining) {
  console.log('Take an umbrella.');
} else {
  console.log('It\'s a nice day.');
}
// Output: It's a nice day.
```

c). Else If

The `else if` statement allows multiple conditions to be checked in a sequence. If the first condition is false, it checks the next condition.

Syntax:

```
if (condition1) {
  // code if condition1 is true
} else if (condition2) {
  // code if condition2 is true
} else {
  // code if all conditions are false
}
```

Example 1:

```
let score = 85;
if (score >= 90) {
  console.log('A');
} else if (score >= 80) {
  console.log('B');
} else {
  console.log('C');
}
// Output: B
```

Example 2:

```
let time = 15;
if (time < 12) {
  console.log('Good Morning!');
} else if (time < 18) {
  console.log('Good Afternoon!');
} else {
  console.log('Good Evening!');
}
// Output: Good Afternoon!
```

d). Switch

The `switch` statement evaluates an expression and executes the corresponding case block if it matches one of the cases.

Syntax:

```
switch (expression) {
  case value1:
    // code if expression === value1
    break;
  case value2:
    // code if expression === value2
    break;
  default:
    // code if no match is found
}
```

Example 1:

```
let day = 3;
switch (day) {
  case 1:
    console.log('Monday');
    break;
  case 2:
    console.log('Tuesday');
    break;
  case 3:
    console.log('Wednesday');
    break;
  default:
    console.log('Invalid day');
}
// Output: Wednesday
```

Example 2:

```
let fruit = 'banana';
switch (fruit) {
  case 'apple':
    console.log('It\'s an apple.');
    break;
  case 'banana':
    console.log('It\'s a banana.');
    break;
  default:
    console.log('Unknown fruit');
}
// Output: It's a banana.
```

f). Loops

i). For

The `for` loop is used for repeating a block of code a specific number of times, based on a condition.

Syntax:

```
for (initialization; condition; increment/decrement) {
  // code to be executed
}
```

Example 1:

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
// Output: 0, 1, 2, 3, 4
```

Example 2:

```
for (let i = 1; i <= 10; i++) {
  console.log(i * 2);
}
// Output: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

ii). For...of

The `for...of` loop is used to iterate over iterable objects like arrays, strings, etc.

Syntax:

```
for (const item of iterable) {
  // code to be executed
}
```

Example 1:

```
let arr = ['apple', 'banana', 'cherry'];
for (const fruit of arr) {
  console.log(fruit);
}
// Output: apple, banana, cherry
```

Example 2:

```
let name = 'John';
for (const letter of name) {
  console.log(letter);
}
// Output: J, o, h, n
```

iii). For...in

The `for...in` loop is used to iterate over the properties (keys) of an object.

Syntax:

```
for (const key in object) {
  // code to be executed
}
```

Example 1:

```
let person = { name: 'Alice', age: 25 };
for (const key in person) {
  console.log(key + ': ' + person[key]);
}
// Output: name: Alice, age: 25
```

Example 2:

```
let car = { make: 'Toyota', model: 'Corolla', year: 2020 };
for (const key in car) {
  console.log(key, car[key]);
}
// Output: make Toyota, model Corolla, year 2020
```

10. Spread

i). How to Copy Properties from One Object into Another Object

The **spread syntax** (`...`) is used to create a shallow copy of an object or merge multiple objects.

Syntax:

```
let newObject = { ...oldObject };
```

Example 1:

```
let person = { name: 'Alice', age: 25 };
let copyPerson = { ...person };
console.log(copyPerson); // Output: { name: 'Alice', age: 25 }
```

Example 2:

```
let car = { make: 'Toyota', model: 'Corolla' };
let copyCar = { ...car, year: 2020 };
console.log(copyCar); // Output: { make: 'Toyota', model: 'Corolla', year:
2020 }
```

ii). How to Copy Elements from One Array into Another Array

The **spread syntax** can also be used to copy elements from one array into another array.

Syntax:

```
let newArray = [...oldArray];
```

Example 1:

```
let arr1 = [1, 2, 3];
let arr2 = [...arr1];
console.log(arr2); // Output: [1, 2, 3]
```

Example 2:

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

11. Rest

a). Rules of Parameter

The **rest syntax** (`...`) is used to collect all remaining arguments into an array. It can only be used as the last parameter in a function.

Syntax:

```
function myFunction(...params) {
  // params is an array containing all passed arguments
}
```

Example 1:

```
function sum(...numbers) {
  return numbers.reduce((a, b) => a + b, 0);
}
console.log(sum(1, 2, 3)); // Output: 6
```

Example 2:

```
function logNames(...names) {
  console.log(names);
}
logNames('Alice', 'Bob', 'Charlie'); // Output: ['Alice', 'Bob', 'Charlie']
```

b). Order of Parameter

The **rest parameter** must always be the last parameter in the function definition.

Syntax:

```
function myFunction(first, second, ...rest) {
  // first and second are regular parameters
  // rest will collect the remaining arguments
}
```

Example 1:

```
function greet(first, second, ...others) {
  console.log(first, second);
  console.log(others);
}
greet('Hello', 'World', 'Alice', 'Bob');
// Output: Hello World, ['Alice', 'Bob']
```

Example 2:

```
function logDetails(name, age, ...details) {
  console.log(name, age);
  console.log(details);
}
logDetails('John', 30, 'Engineer', 'New York');
// Output: John 30, ['Engineer', 'New York']
```

12. Destructuring

a). Object Destructuring

Object destructuring allows you to unpack values from objects into distinct variables.

Syntax:

```
let { property1, property2 } = object;
```

Example 1:

```
let person = { name: 'Alice', age: 25 };
let { name, age } = person;
console.log(name); // Output: Alice
console.log(age);  // Output: 25
```

Example 2:

```
let car = { make: 'Toyota', model: 'Corolla', year: 2020 };
let { make, model } = car;
console.log(make); // Output: Toyota
console.log(model); // Output: Corolla
```

b). Array Destructuring

Array destructuring allows you to unpack values from arrays into distinct variables.

Syntax:

```
let [element1, element2] = array;
```

Example 1:

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

Example 2:

```
let fruits = ['apple', 'banana', 'cherry'];
let [fruit1, fruit2] = fruits;
console.log(fruit1); // Output: apple
console.log(fruit2); // Output: banana
```

13. Scopes

a). Global Scope

A variable has **global scope** if it is declared outside of any function or block, meaning it is accessible anywhere in the code.

Example 1:

```
let globalVar = 'I am global';
function displayGlobal() {
  console.log(globalVar); // Output: I am global
}
displayGlobal();
```

Example 2:

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

b). Function Scope

A variable has **function scope** if it is declared inside a function. It is only accessible within that function.

Example 1:

```
function myFunction() {
  let localVar = 'I am local';
  console.log(localVar); // Output: I am local
}
myFunction();
console.log(localVar); // Error: localVar is not defined
```

Example 2:

```
function add() {
  var sum = 10 + 5;
  console.log(sum); // Output: 15
}
add();
console.log(sum); // Error: sum is not defined
```

c). Block Scope

Variables declared with `let` and `const` have **block scope**. They are only accessible within the block (e.g., within curly braces `{}`).

Example 1:

```
if (true) {
  let blockVar = 'Inside block';
  console.log(blockVar); // Output: Inside block
}
console.log(blockVar); // Error: blockVar is not defined
```

Example 2:

```
for (let i = 0; i < 3; i++) {
  console.log(i); // Output: 0, 1, 2
}
console.log(i); // Error: i is not defined
```

d). Lexical Scope

Lexical scope refers to the location where a variable is declared within the code and determines its accessibility.

Example 1:

```
let globalVar = 'I am global';
function outer() {
  let outerVar = 'I am outer';
  function inner() {
    console.log(globalVar); // Accessible (global scope)
    console.log(outerVar);  // Accessible (lexical scope)
  }
  inner();
}
outer();
```

Example 2:

```
function outer() {
  let outerVar = 'Outer variable';
  function inner() {
    console.log(outerVar); // Output: Outer variable
  }
  inner();
}
outer();
```

e). Var

Variables declared with **`var`** have **function scope** (if declared inside a function) or **global scope** (if declared outside any function).

Example 1:

```
var x = 10;
if (true) {
  var x = 20; // Reassigns the global variable
}
console.log(x); // Output: 20
```

Example 2:

```
function myFunc() {
  var x = 30;
  console.log(x); // Output: 30
}
myFunc();
```

f). Let

Variables declared with **`let`** have **block scope**.

Example 1:

```
let x = 10;
if (true) {
  let x = 20; // Declares a new variable in the block
  console.log(x); // Output: 20
}
console.log(x); // Output: 10
```

Example 2:

```
let a = 100;
if (true) {
  let a = 200; // Block scoped
  console.log(a); // Output: 200
}
console.log(a); // Output: 100
```

g). Const

Variables declared with `const` have **block scope** and cannot be reassigned after initialization.

Example 1:

```
const x = 10;
// x = 20; // Error: Assignment to constant variable
```

Example 2:

```
const person = { name: 'Alice' };
person.name = 'Bob'; // Allowed (modifies the object)
console.log(person); // Output: { name: 'Bob' }
```

h). Difference Between var, let, and const

- `var`: Function-scoped, can be re-declared and reassigned.
- `let`: Block-scoped, can be reassigned but cannot be redeclared in the same scope.
- `const`: Block-scoped, cannot be reassigned or re-declared, must be initialized when declared.
-

14. `this` Keyword

a). Browser Context

- **Definition**: In the browser context, `this` refers to the global object (usually `window`), or to the object invoking a method.

- **Syntax**:
    - `this` refers to the current context, which is determined by how the function is called.

i). Arrow Function

Definition: In an arrow function, `this` is lexically inherited from the surrounding context (does not have its own `this`).

Syntax:

```
const func = () => { console.log(this); }
```

Example 1:

```
const obj = {
  name: 'Alice',
  greet: () => {
    console.log(this.name); // Output: undefined (inherited from global
context)
  }
};
obj.greet();
```

Example 2:

```
const obj = {
  name: 'Bob',
  greet: function() {
    const inner = () => {
      console.log(this.name); // Output: Bob (inherited from outer
function's `this`)
    };
    inner();
  }
};
obj.greet();
```

ii). Named Function

Definition: In a regular function (named), this refers to the object invoking the method.

Syntax:

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

Example 1:

```
const obj = {
  name: 'Charlie',
  greet: function() {
    console.log(this.name); // Output: Charlie
  }
};
obj.greet();
```

Example 2:

```
function showName() {
  console.log(this.name); // Output: undefined (global `this` in browser)
}
showName();
```

b). Node Context

> **Definition**: In **Node.js**, this refers to the module context (module.exports) or global object depending on how the function is invoked.

i). Arrow Function

> **Definition**: Arrow functions in Node inherit this from the surrounding lexical context, just like in the browser.

Example 1:

```
const obj = {
  name: 'Node',
  greet: () => {
    console.log(this.name); // Output: undefined (lexical this, from global context)
  }
};
obj.greet();
```

Example 2:

```
const obj = {
  name: 'Express',
  greet: function() {
    const inner = () => {
      console.log(this.name); // Output: Express (inherited from outer function's `this`)
    };
    inner();
  }
};
obj.greet();
```

ii). Named Function

> **Definition**: In a **named function** in Node.js, this can refer to the global object or the object invoking the function, depending on the context.

Example 1:

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

Example 2:

```
function showContext() {
  console.log(this); // Output: global object in Node.js (not `window` like in the browser)
}
showContext();
```

15. Call

a). What is Call

Definition: The call() method allows you to invoke a function with a specific this value and arguments passed individually.

Syntax:

```
functionName.call(thisContext, arg1, arg2, ...);
```

b). Purpose

Purpose: It is used to invoke a function with a specific context (this) and can pass any number of arguments.

c). How to Work with It

You use call() to invoke a function and specify its this context and arguments.

Example 1:

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

const person = { name: 'Alice' };
greet.call(person); // Output: Hello, Alice
```

Example 2:

```
function sum(a, b) {
  console.log(a + b);
}
sum.call(null, 5, 3); // Output: 8
```

16. Apply

a). What is Apply

Definition: The apply() method is similar to call(), but it takes an array or array-like object of arguments instead of individual arguments.

Syntax:

```
functionName.apply(thisContext, [arg1, arg2, ...]);
```

b). Purpose

> **Purpose**: It is used to invoke a function with a specific this context and an array of arguments.

c). How to Work with It

> You use apply() when you want to pass an array of arguments to the function.

Example 1:

```
function greet(message) {
  console.log(message + ', ' + this.name);
}

const person = { name: 'Bob' };
greet.apply(person, ['Hello']); // Output: Hello, Bob
```

Example 2:

```
function sum(a, b, c) {
  console.log(a + b + c);
}
```

```
sum.apply(null, [1, 2, 3]); // Output: 6
```

17. Bind

a). What is Bind

Definition: The bind() method creates a new function that, when invoked, has a specific this value and any number of arguments pre-set.

Syntax:

```
let boundFunction = functionName.bind(thisContext, arg1, arg2, ...);
```

b). Purpose

> **Purpose**: It is used to set a fixed this context for a function, and it returns a new function with that this context.

c). How to Work with It

> You use bind() when you need to create a new function that can be invoked later, with a fixed this value and pre-defined arguments.

Example 1:

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

Example 2:

```
function add(a, b) {
  console.log(a + b);
}

const addFive = add.bind(null, 5);
addFive(3); // Output: 8
```

18. Difference Between call(), apply(), and bind()

Key Comparison Table

Feature	`call()`	`apply()`	`bind()`
Execution	Invokes the function immediately	Invokes the function immediately	Returns a new function (no execution)
Argument Format	Individual arguments	Arguments as an array	Arguments can be partially applied
Return Value	Result of the function	Result of the function	A new bound function
Use Case	Immediate execution with `this`	Immediate execution with `this`	Reusable functions with specific `this`

a). Call

Definition: The call() method invokes a function immediately with a specified this context and arguments passed individually.

Syntax:

```
functionName.call(thisContext, arg1, arg2, ...);
```

Example 1:

```
function greet(message) {
  console.log(message + ', ' + this.name);
}

const person = { name: 'Alice' };
greet.call(person, 'Hello'); // Output: Hello, Alice
```

Example 2:

```
function sum(a, b) {
```

```
  console.log(a + b);
}
sum.call(null, 5, 3); // Output: 8
```

2. Apply

- **Definition**: The apply() method is similar to call(), but it takes an array (or array-like object) of arguments instead of individual arguments.

Syntax:
javascript
CopyEdit
functionName.apply(thisContext, [arg1, arg2, ...]);

-

Example 1:
javascript
CopyEdit
function greet(message) {

  console.log(message + ', ' + this.name);

}

const person = { name: 'Bob' };

greet.apply(person, ['Hello']); // Output: Hello, Bob

-

Example 2:
javascript
CopyEdit
function sum(a, b, c) {

  console.log(a + b + c);

}

```
sum.apply(null, [1, 2, 3]); // Output: 6
```

-

3. Bind

- **Definition**: The bind() method creates a new function with a specified this context and pre-defined arguments, but does not invoke the function immediately.

Syntax:
javascript
CopyEdit
```
let boundFunction = functionName.bind(thisContext, arg1, arg2, ...);
```

-

Example 1:
javascript
CopyEdit
```
function greet() {

  console.log('Hello, ' + this.name);

}



const person = { name: 'Charlie' };

const greetPerson = greet.bind(person);

greetPerson(); // Output: Hello, Charlie
```

-

Example 2:
javascript
CopyEdit
```
function add(a, b) {

  console.log(a + b);

}



const addFive = add.bind(null, 5);
```

```
addFive(3); // Output: 8
```

-

Closures

1. What is Closures

- **Definition**: A closure is a function that remembers and has access to its lexical scope, even when the function is executed outside that scope.

2. How to Create a Closure

- **Definition**: Closures are created when a function is defined inside another function, and the inner function accesses variables from the outer function.

3. Syntax

javascript

CopyEdit

```
function outerFunction() {
  let outerVar = 'I am from outer function';

  function innerFunction() {
    console.log(outerVar); // Closure
  }
  return innerFunction;
}
```

4. Purpose

- **Purpose**: Closures are useful for maintaining state, data encapsulation, and implementing function factories.

Example 1:
javascript
CopyEdit

```
function outer() {
  let counter = 0;
  function increment() {
    counter++;
    console.log(counter);
  }
  return increment;
}
const counterClosure = outer();
counterClosure(); // Output: 1
counterClosure(); // Output: 2
```

-

Example 2:
javascript
CopyEdit

```
function greeting(message) {
  return function(name) {
    console.log(message + ', ' + name);
  }
}
const greetHello = greeting('Hello');
greetHello('Alice'); // Output: Hello, Alice
```

-

Promises

1. Creation of Promise

- **Definition**: A promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Syntax:
javascript
CopyEdit
```
const promise = new Promise((resolve, reject) => {

  // Asynchronous code here

});
```

-

Example 1:
javascript
CopyEdit
```
const promise = new Promise((resolve, reject) => {

  let success = true;

  if (success) {

    resolve('Operation succeeded!');

  } else {

    reject('Operation failed!');

  }

});
```

-

Example 2:
javascript
CopyEdit
```
const promise = new Promise((resolve, reject) => {

  let data = { name: 'Alice', age: 25 };

  if (data) {
```

```
    resolve(data);
  } else {
    reject('Data not found');
  }
});
```

-

2. Resolve

- **Definition**: resolve() is called when the asynchronous operation is successful, and it provides the result of the operation.

Syntax:
javascript
CopyEdit
```
resolve(value);
```

-

Example 1:
javascript
CopyEdit
```
const promise = new Promise((resolve) => {
  resolve('Data received');
});

promise.then(data => console.log(data)); // Output: Data received
```

-

Example 2:
javascript
CopyEdit
```
const promise = new Promise((resolve) => {
  resolve('Task completed');
});

promise.then(result => console.log(result)); // Output: Task completed
```

-

3. Reject

- **Definition**: reject() is called when the asynchronous operation fails, and it provides the error message or reason for failure.

Syntax:
javascript
CopyEdit
reject(error);

-

Example 1:
javascript
CopyEdit
```
const promise = new Promise((_, reject) => {

  reject('Error occurred');

});

promise.catch(error => console.log(error)); // Output: Error occurred
```

-

Example 2:
javascript
CopyEdit
```
const promise = new Promise((_, reject) => {

  reject('Something went wrong');

});

promise.catch(error => console.log(error)); // Output: Something went wrong
```

-

4. States of Promise

- **Definition**: A promise can be in one of the following states:
    - **Pending**: Initial state, neither fulfilled nor rejected.
    - **Fulfilled**: The asynchronous operation completed successfully.
    - **Rejected**: The asynchronous operation failed.

Example 1:
javascript
CopyEdit

```
const promise = new Promise((resolve, reject) => {

  let success = true;

  if (success) {

    resolve('Operation successful');

  } else {

    reject('Operation failed');

  }

});

promise.then(result => console.log(result)); // Output: Operation successful

promise.catch(error => console.log(error)); // Output: Operation failed
```

-

Example 2:
javascript
CopyEdit

```
const promise = new Promise((resolve, reject) => {

  let success = false;

  if (success) {

    resolve('Data fetched');

  } else {

    reject('Data fetch failed');

  }

});

promise.then(result => console.log(result)); // Output: Data fetched

promise.catch(error => console.log(error)); // Output: Data fetch failed
```

-

5. Accessing the Data from Promise

Then and Catch

Syntax:
javascript
CopyEdit
promise.then(result => {

  // handle success

}).catch(error => {

  // handle error

});

-

Example 1:
javascript
CopyEdit
const promise = new Promise((resolve, reject) => {

  resolve('Success!');

});

promise.then(result => {

  console.log(result); // Output: Success!

}).catch(error => {

  console.log(error);

});

-

Example 2:
javascript
CopyEdit
const promise = new Promise((_, reject) => {

```
  reject('Failed to fetch data');
});

promise.then(result => {
  console.log(result);
}).catch(error => {
  console.log(error); // Output: Failed to fetch data
});
```

-

Async/Await

- **Definition**: async/await provides a more readable and synchronous-like way of working with promises.

Syntax:
javascript
CopyEdit

```
async function fetchData() {
  try {
    const result = await promise;
    console.log(result);
  } catch (error) {
    console.log(error);
  }
}
```

-

Example 1:
javascript
CopyEdit

```
const fetchData = async () => {
```

```
  const promise = new Promise((resolve, reject) => {
    resolve('Data received');
  });

  try {
    const result = await promise;
    console.log(result); // Output: Data received
  } catch (error) {
    console.log(error);
  }
};

fetchData();
```

-

Example 2:
javascript
CopyEdit

```
async function fetchData() {
  const promise = new Promise((_, reject) => {
    reject('Failed to fetch data');
  });

  try {
    const result = await promise;
    console.log(result);
  } catch (error) {
```

```
    console.log(error); // Output: Failed to fetch data
  }
}
fetchData();
```

-

Try/Catch with Async/Await

Syntax:
javascript
CopyEdit

```
async function fetchData() {
  try {
    const result = await promise;
    console.log(result);
  } catch (error) {
    console.log(error);
  }
}
```

-

Example 1:
javascript
CopyEdit

```
async function fetchData() {
  const promise = new Promise((resolve, reject) => {
    resolve('Task completed');
  });

  try {
```

```
    const result = await promise;
    console.log(result); // Output: Task completed
  } catch (error) {
    console.log(error);
  }
}
fetchData();
```

-

Example 2:
javascript
CopyEdit

```
async function fetchData() {
  const promise = new Promise((_, reject) => {
    reject('Error encountered');
  });

  try {
    const result = await promise;
    console.log(result);
  } catch (error) {
    console.log(error); // Output: Error encountered
  }
}
```

- fetchData();

DOM and DOM Manipulation

1. What is DOM

- **Definition**: The Document Object Model (DOM) represents the structure of an HTML document as an object-oriented tree structure, where each node is an object representing part of the page (e.g., elements, attributes, and text).
- **Purpose**: It allows programs and scripts to dynamically access and update the content, structure, and style of web pages.

Example 1:
javascript
CopyEdit
// Accessing an element using DOM

document.getElementById('example').innerText = 'Hello, DOM!';

-

Example 2:
javascript
CopyEdit
const divElement = document.createElement('div');

divElement.textContent = 'This is a new div!';

document.body.appendChild(divElement);

-

2. DOM Objects

- **Definition**: DOM objects are the elements, attributes, and text nodes within an HTML document that are represented as objects in the DOM.

Example 1:
javascript
CopyEdit
const titleElement = document.getElementById('title');

console.log(titleElement); // DOM object representing the title element

-

Example 2:
javascript
CopyEdit
const paragraphs = document.getElementsByClassName('para');

console.log(paragraphs); // DOM collection of all elements with class 'para'

-

3. How to Access the DOM Object

- **Definition**: DOM objects can be accessed using methods like getElementById(), getElementsByClassName(), and querySelector().

Example 1:
javascript
CopyEdit
const heading = document.getElementById('heading');

console.log(heading); // Accessing element by ID

-

Example 2:
javascript
CopyEdit
const listItems = document.querySelectorAll('.list-item');

console.log(listItems); // Accessing all elements with class 'list-item'

-

4. How to Manipulate HTML with DOM

- **Definition**: The DOM provides methods for modifying HTML elements, adding new content, changing styles, and handling events.

Example 1:
javascript
CopyEdit
const header = document.getElementById('header');

header.innerHTML = '<h1>New Header Content</h1>'; // Manipulating inner HTML

-

Example 2:
javascript
CopyEdit
const button = document.querySelector('button');

button.style.backgroundColor = 'green'; // Changing styles via DOM

-

DOM Methods

5. getElementById

- **Definition**: The getElementById() method returns a reference to the first object with the specified id.

Syntax:
javascript
CopyEdit
document.getElementById('id');

-

Example 1:
javascript
CopyEdit
const element = document.getElementById('uniqueId');

console.log(element); // Access element with id 'uniqueId'

-

Example 2:
javascript
CopyEdit
const button = document.getElementById('submitButton');

button.style.color = 'red'; // Changing button color via id

-

6. getElementsByClassName

- **Definition**: The getElementsByClassName() method returns a live HTMLCollection of all elements with the specified class name.

Syntax:
javascript
CopyEdit
document.getElementsByClassName('className');

-

Example 1:
javascript
CopyEdit
const paragraphs = document.getElementsByClassName('para');

console.log(paragraphs); // Access all elements with class 'para'

-

Example 2:
javascript
CopyEdit
const listItems = document.getElementsByClassName('list');

listItems[0].style.color = 'blue'; // Changing the first list item color

-

7. querySelector

- **Definition**: The querySelector() method returns the first element that matches a specified CSS selector.

Syntax:
javascript
CopyEdit
document.querySelector('selector');

-

Example 1:
javascript
CopyEdit
const firstButton = document.querySelector('button');

firstButton.style.fontSize = '20px'; // Styling the first button

-

Example 2:
javascript
CopyEdit
const header = document.querySelector('#header');

header.textContent = 'Updated Header'; // Access and update text of header

-

8. innerText

- **Definition**: innerText is a property that gets or sets the text content of an element, excluding HTML tags.

Syntax:
javascript
CopyEdit
element.innerText;

element.innerText = 'new text';

-

Example 1:
javascript
CopyEdit
const paragraph = document.getElementById('para');

paragraph.innerText = 'New paragraph content'; // Updating text content

-

Example 2:
javascript
CopyEdit
const header = document.getElementById('header');

console.log(header.innerText); // Logging text content of header

-

9. innerHTML

- **Definition**: innerHTML is a property that gets or sets the HTML content inside an element.

Syntax:
javascript
CopyEdit
element.innerHTML;

element.innerHTML = '<h1>New HTML content</h1>';

-

Example 1:
javascript
CopyEdit
```
const contentDiv = document.getElementById('content');

contentDiv.innerHTML = '<p>Updated content with HTML</p>'; // Updating inner HTML
```

-

Example 2:
javascript
CopyEdit
```
const footer = document.getElementById('footer');

console.log(footer.innerHTML); // Logging inner HTML content
```

-

10. append

- **Definition**: The append() method adds a new element or text to the end of an element's content.

Syntax:
javascript
CopyEdit
```
parentElement.append(childElement);
```

-

Example 1:
javascript
CopyEdit
```
const div = document.createElement('div');

div.textContent = 'Appended Div';

document.body.append(div); // Append div to the body
```

-

Example 2:
javascript
CopyEdit
```
const paragraph = document.createElement('p');

paragraph.textContent = 'This is a new paragraph';
```

document.getElementById('container').append(paragraph); // Append paragraph to a container

-

11. appendChild

- **Definition**: The appendChild() method appends a child element to the specified parent node.

Syntax:
javascript
CopyEdit
parentElement.appendChild(childElement);

-

Example 1:
javascript
CopyEdit
const li = document.createElement('li');

li.textContent = 'New List Item';

document.getElementById('list').appendChild(li); // Append list item to an existing list

-

Example 2:
javascript
CopyEdit
const span = document.createElement('span');

span.textContent = 'Appended Text';

document.getElementById('textContainer').appendChild(span); // Append span to container

-

CreateElement

1. How to Create a DOM Element

- **Definition**: The createElement() method creates a new HTML element node.

Syntax:
javascript
CopyEdit
document.createElement('elementName');

-

Example 1:
javascript
CopyEdit
const divElement = document.createElement('div');

divElement.textContent = 'Created new div element';

document.body.appendChild(divElement); // Adding created div to the body

-

Example 2:
javascript
CopyEdit
const h2 = document.createElement('h2');

h2.textContent = 'Dynamic Header';

document.body.appendChild(h2); // Adding dynamic header to the body

-

2. How to Add Content in It

- **Definition**: After creating a DOM element, content can be added using properties like textContent or innerHTML.

Example 1:
javascript
CopyEdit
const pElement = document.createElement('p');

pElement.textContent = 'This is a dynamically created paragraph';

document.body.appendChild(pElement); // Adding paragraph to the body

-

Example 2:
javascript

CopyEdit
```
const anchor = document.createElement('a');

anchor.innerHTML = 'Click Here';

anchor.href = 'https://example.com';

document.body.appendChild(anchor); // Adding anchor to the body
```

-

3. How to Add DOM Element in the DOM

- **Definition**: A newly created DOM element is added to the DOM tree using appendChild(), insertBefore(), or other methods.

Example 1:
javascript
CopyEdit
```
const newDiv = document.createElement('div');

newDiv.textContent = 'New dynamically added div';

document.getElementById('parent').appendChild(newDiv); // Appending to specific parent
```

-

Example 2:
javascript
CopyEdit
```
const newSection = document.createElement('section');

newSection.innerHTML = '<h2>New Section</h2>';

document.body.appendChild(newSection); // Appending section to the body
```

-

4. How to Remove DOM Element

- **Definition**: DOM elements can be removed using removeChild() or remove() methods.

Example 1:
javascript
CopyEdit
```
const divToRemove = document.getElementById('removeMe');
```

```
divToRemove.remove(); // Removing an element by calling remove()
```

-

Example 2:
javascript
CopyEdit
```
const parent = document.getElementById('parent');

const child = document.getElementById('child');

parent.removeChild(child); // Removing a child element from its parent
```

-

Fetch API

1. Get Request

- **Definition**: The Fetch API provides an easy-to-use interface for making HTTP requests. The GET request is used to retrieve data from the server.

Syntax:
javascript
CopyEdit
```
fetch(url)

  .then(response => response.json())

  .then(data => console.log(data))

  .catch(error => console.error('Error:', error));
```

-

Example 1:
javascript
CopyEdit
```
fetch('https://jsonplaceholder.typicode.com/posts')

  .then(response => response.json())

  .then(data => console.log(data)) // Fetching posts data

  .catch(error => console.error('Error:', error));
```

-

Example 2:
javascript
CopyEdit
```
fetch('https://api.example.com/users')

  .then(response => response.json())

  .then(data => console.log(data)) // Fetching user data

  .catch(error => console.error('Error:', error));
```

-

OOPs (Object-Oriented Programming)

1. Class

- **Definition**: A class is a blueprint for creating objects, providing initial values for state (member variables) and implementations of behavior (methods or functions).

Syntax:
javascript
CopyEdit
```
class ClassName {

  constructor() {

    // constructor body

  }

}
```

-

Example 1:
javascript
CopyEdit
```
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.`);
  }
}

const person1 = new Person('Alice', 25);
person1.greet(); // Hello, my name is Alice and I am 25 years old.
```

-

Example 2:
javascript
CopyEdit

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

  start() {
    console.log(`Starting ${this.make} ${this.model}`);
  }
}

const car1 = new Car('Toyota', 'Corolla');
```

car1.start(); // Starting Toyota Corolla

-

2. Object

- **Definition**: An object is an instance of a class. It holds properties (data) and methods (functions) defined in the class.

Syntax:
javascript
CopyEdit
const objectName = new ClassName();

-

Example 1:
javascript
CopyEdit
const person = new Person('John', 30);

person.greet(); // Hello, my name is John and I am 30 years old.

-

Example 2:
javascript
CopyEdit
const car = new Car('Honda', 'Civic');

car.start(); // Starting Honda Civic

-

3. Constructor

- **Definition**: The constructor is a special method in a class that is called when a new object of the class is created. It is used to initialize properties of the object.

Syntax:
javascript
CopyEdit
constructor(parameters) {

  // initialization code

}

-

Example 1:
javascript
CopyEdit

```
class Animal {

  constructor(name, sound) {

    this.name = name;

    this.sound = sound;

  }



  makeSound() {

    console.log(`${this.name} says ${this.sound}`);

  }

}



const dog = new Animal('Dog', 'Woof');

dog.makeSound(); // Dog says Woof
```

-

Example 2:
javascript
CopyEdit

```
class Book {

  constructor(title, author) {

    this.title = title;

    this.author = author;

  }
```

```
  getBookInfo() {
    console.log(`Book: ${this.title} by ${this.author}`);
  }
}

const book1 = new Book('1984', 'George Orwell');
book1.getBookInfo(); // Book: 1984 by George Orwell
```

-

4. Inheritance

- **Definition**: Inheritance is a mechanism where one class can inherit properties and methods from another class, allowing for code reuse.

Syntax:
javascript
CopyEdit
```
class ChildClass extends ParentClass {
  // additional methods or properties
}
```

-

Example 1:
javascript
CopyEdit
```
class Animal {
  constructor(name) {
    this.name = name;
  }

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

```
  }
}

class Dog extends Animal {
  constructor(name) {
    super(name); // Calling the parent class constructor
  }

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

const dog = new Dog('Buddy');
dog.speak(); // Buddy barks.
```

-

Example 2:
javascript
CopyEdit

```
class Shape {
  constructor(type) {
    this.type = type;
  }

  describe() {
    console.log(`This is a ${this.type}`);
```

```
  }
}


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


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


const circle = new Circle(5);
circle.describe(); // This is a Circle
console.log(circle.area()); // 78.53981633974483
```

-

5. Polymorphism with Overriding

> **Definition**: Polymorphism allows a subclass to provide a specific implementation of a method that is already defined in its parent class, a concept known as method overriding.

Syntax:
javascript
CopyEdit
```
class ChildClass extends ParentClass {
  methodName() {
```

```
    // Overriding parent method
  }
}
```

Example 1:
javascript
CopyEdit

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

class Dog extends Animal {
  speak() {
    console.log('Dog barks');
  }
}

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

Example 2:
javascript
CopyEdit

```
class Vehicle {
  start() {
```

```
    console.log('Vehicle is starting');
  }
}

class Car extends Vehicle {
  start() {
    console.log('Car is starting');
  }
}

const car = new Car();
car.start(); // Car is starting
```

Module Concept in JS

1. What is Module Concept

> **Definition**: In JavaScript, modules are reusable pieces of code that can be imported and exported to avoid polluting the global namespace. Modules help in organizing code into smaller, manageable units.

Example 1:
javascript
CopyEdit

```
// In math.js (module)

export const add = (a, b) => a + b;

export const subtract = (a, b) => a - b;
```

javascript
CopyEdit

```
// In app.js

import { add, subtract } from './math.js';

console.log(add(2, 3)); // 5

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

Example 2:
javascript
CopyEdit

```
// In greetings.js

export const greet = name => `Hello, ${name}!`;
```

javascript
CopyEdit

```
// In app.js

import { greet } from './greetings.js';

console.log(greet('Alice')); // Hello, Alice!
```

2. Why We Use It

> **Definition**: Modules are used to split code into smaller parts, making it easier to maintain, debug, and test. It also helps in code reuse and modularity.

Example 1:
javascript
CopyEdit

```
// utils.js

export function formatDate(date) {

  return date.toISOString().split('T')[0];

}
```

javascript
CopyEdit
// app.js

import { formatDate } from './utils.js';

console.log(formatDate(new Date())); // 2025-01-25

Example 2:
javascript
CopyEdit
// calculate.js

export const multiply = (a, b) => a * b;

javascript
CopyEdit
// app.js

import { multiply } from './calculate.js';

console.log(multiply(3, 4)); // 12

3. Named Export

> **Definition**: Named exports allow you to export multiple values from a module by their names, which must be imported using the same names.

Syntax:
javascript
CopyEdit
export const name = value;

Example 1:
javascript
CopyEdit
// module.js

export const square = x => x * x;

export const cube = x => x * x * x;

javascript
CopyEdit
```
// app.js

import { square, cube } from './module.js';

console.log(square(3)); // 9

console.log(cube(3)); // 27
```

Example 2:
javascript
CopyEdit
```
// util.js

export const double = x => x * 2;
```

javascript
CopyEdit
```
// app.js

import { double } from './util.js';

console.log(double(4)); // 8
```

4. Default Export

> **Definition**: Default exports allow you to export a single value from a module. The value can be imported without specifying the name.

Syntax:
javascript
CopyEdit
```
export default value;
```

Example 1:
javascript
CopyEdit
```
// math.js

export default function add(a, b) {
```

```
  return a + b;

}
```

```
javascript
CopyEdit
// app.js

import add from './math.js';

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

Example 2:

```
javascript
CopyEdit
// logger.js

export default function log(message) {

  console.log(message);

}
```

```
javascript
CopyEdit
// app.js

import log from './logger.js';

log('Hello, World!'); // Hello, World!
```

5. Importing Named Export

> **Definition**: Named exports are imported by specifying the exact names of the variables or functions you want to use.

Syntax:

```
javascript
CopyEdit
import { name1, name2 } from 'module';
```

Example 1:
javascript
CopyEdit
// math.js

export const sum = (a, b) => a + b;

export const subtract = (a, b) => a - b;

javascript
CopyEdit
// app.js

import { sum, subtract } from './math.js';

console.log(sum(1, 2)); // 3

console.log(subtract(5, 3)); // 2

Example 2:
javascript
CopyEdit
// colors.js

export const red = '#FF0000';

export const blue = '#0000FF';

javascript
CopyEdit
// app.js

import { red, blue } from './colors.js';

console.log(red); // #FF0000

console.log(blue); // #0000FF

6. Importing Default Export

Definition: A default export is imported without the need for curly braces.

Syntax:
javascript
CopyEdit
import variable from 'module';

Example 1:
javascript
CopyEdit
// math.js

export default function multiply(a, b) {

  return a * b;

}

javascript
CopyEdit
// app.js

import multiply from './math.js';

console.log(multiply(2, 3)); // 6

Example 2:
javascript
CopyEdit
// greeting.js

export default function greet(name) {

  return `Hello, ${name}!`;

}

javascript
CopyEdit
// app.js

import greet from './greeting.js';

console.log(greet('Alice')); // Hello, Alice!