### Console.log

**Explanation:** console.log is used to print output to the console. It's helpful for debugging by allowing you to inspect variable values, execution flow, or messages.

**Syntax:**

console.log(value);

**Example:**

const name = 'John';

console.log(name); // Output: John

### Variables: let, const, var

#### var

**Explanation:** var is function-scoped or globally-scoped and is hoisted to the top of its scope with an initial value of undefined. This can lead to unexpected behavior due to lack of block scope.

**Syntax:**

var variableName = value;

**Example:**

function example() {

var x = 10;

if (true) {

var x = 20; // Same variable, function-scoped

console.log(x); // Output: 20

}

console.log(x); // Output: 20

}

#### let

**Explanation:** let is block-scoped, meaning it is limited to the block in which it is defined. It is also hoisted but not initialized, which means it cannot be accessed before its declaration (temporal dead zone).

**Syntax:**

let variableName = value;

**Example:**

function example() {

let x = 10;

if (true) {

let x = 20; // Different variable, block-scoped

console.log(x); // Output: 20

}

console.log(x); // Output: 10

}

#### const

**Explanation:** const is block-scoped like let, but it is used for variables whose value should not be reassigned. It must be initialized at the time of declaration.

**Syntax:**

const variableName = value;

**Example:**

const pi = 3.14;

// pi = 3.1415; // Error: Assignment to constant variable.

### Data Types in JavaScript

#### String

**Explanation:** Represents a sequence of characters. Can be enclosed in single quotes, double quotes, or backticks (for template literals).

**Syntax:**

const str = 'Hello, World!';

**Example:**

const greeting = "Hello, " + "World!"; // Output: Hello, World!

#### Number

**Explanation:** Represents both integer and floating-point numbers. JavaScript numbers are limited to the range from -(2^53 - 1) to 2^53 - 1.

**Syntax:**

const num = 123;

**Example:**

const amount = 25; // Output: 25

#### BigInt

**Explanation:** Represents integers with arbitrary precision. Use n suffix to denote a BigInt.

**Syntax:**

const bigNum = 1234567890123456789012345678901234567890n;

**Example:**

const bigNumber = 1234567890123456789012345678901234567890n;

#### Boolean

**Explanation:** Represents a value that is either true or false.

**Syntax:**

const isTrue = true;

**Example:**

javascript

Copy code

const isActive = Boolean(1); // Output: true

#### Null

**Explanation:** Represents the intentional absence of any value.

**Syntax:**

const emptyValue = null;

**Example:**

const noValue = null;

#### Undefined

**Explanation:** Represents a variable that has been declared but not assigned a value.

**Syntax:**

let uninitialized;

**Example:**

let value;

console.log(value); // Output: undefined

#### Symbol

**Explanation:** A unique and immutable primitive value that can be used as an object property key.

**Syntax:**

const sym = Symbol('description');

**Example:**

const uniqueSymbol = Symbol('description');

#### Object

**Explanation:** A collection of key-value pairs where keys are strings (or symbols) and values can be any data type.

**Syntax:**

const obj = { key: 'value' };

**Example:**

const person = { name: 'John', age: 25 };

**Type Checking:**

typeof variableName;

**Example:**

console.log(typeof person); // Output: object

### Type Conversion

**String to Number:**

const num = Number('123'); // Output: 123

**Number to Boolean:**

const bool = Boolean(0); // Output: false

**Number to String:**

const str = String(123); // Output: '123'

### Comparison Operators

**Explanation:** Used to compare values and determine their relationship.

**Syntax:**

* Greater than: a > b
* Less than: a < b
* Greater than or equal to: a >= b
* Less than or equal to: a <= b
* Not equal to: a != b
* Equal to: a == b (allows type coercion)
* Strictly equal to: a === b (no type coercion)

**Examples:**

console.log("2" == 2); // Output: true

console.log("2" === 2); // Output: false

### String Concatenation

**Explanation:** Combine two or more strings into one.

**Syntax:**

string1 + string2;

**Example:**

const firstName = 'John';

const lastName = 'Doe';

const fullName = firstName + ' ' + lastName; // Output: 'John Doe'

### Logical Operators

**Purpose:** Used to perform logical operations and return a Boolean value (true or false).

**Operators:**

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

operand1 && operand2;

**Example:**

console.log(true && false); // Output: false

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

operand1 || operand2;

**Example:**

console.log(true || false); // Output: true

* **! (Logical NOT):** Returns true if the operand is false and vice versa.

!operand;

**Example:**

console.log(!true); // Output: false

### Arithmetic Operators

**Purpose:** Used to perform mathematical operations.

**Operators:**

* **+ (Addition):** Adds two operands.

operand1 + operand2;

**Example:**

console.log(5 + 3); // Output: 8

* **- (Subtraction):** Subtracts the second operand from the first.

operand1 - operand2;

**Example:**

console.log(5 - 3); // Output: 2

* **\* (Multiplication):** Multiplies two operands.

operand1 \* operand2;

**Example:**

console.log(5 \* 3); // Output: 15

* **/ (Division):** Divides the first operand by the second.

operand1 / operand2;

**Example:**

console.log(6 / 3); // Output: 2

* **% (Modulo):** Returns the remainder of the division of the first operand by the second.

operand1 % operand2;

**Example:**

console.log(5 % 3); // Output: 2

* **++ (Increment):** Increases the value of the operand by 1.

operand++;

**Example:**

let x = 5;

x++;

console.log(x); // Output: 6

* **-- (Decrement):** Decreases the value of the operand by 1.

operand--;

**Example:**

let y = 5;

y--;

console.log(y); // Output: 4

### Non-Primitive Data Types in JavaScript

1. **Object**

**Explanation:** Objects are collections of key-value pairs. Keys are usually strings (or symbols) and values can be any data type.

**Syntax:**

let objectName = {

key1: value1,

key2: value2,

// more key-value pairs

};

**Example:**

let data = {

"name": "nirajan",

"age": 20

};

1. **Array**

**Explanation:** Arrays are ordered collections of values. Values can be of any data type and are accessed by their index.

**Syntax:**

let arrayName = [value1, value2, value3, ...];

**Example:**

let a = ["nirajan", "kirajan", "birajan"];

1. **Function**

**Explanation:** Functions are blocks of code designed to perform a particular task. They can be invoked (called) to execute their code.

**Syntax:**

function functionName(parameters) {

// code to be executed

}

**Example:**

function outer() {

console.log("hi");

}

outer(); // Output: hi

### String Methods and Operations

1. **Concatenation**
   * **Explanation:** Combines two or more strings into one.
   * **Syntax:**

string1 + string2;

* + **Returns:** A new string that is the concatenation of the original strings.
  + **Example:**

let a = ' My name is nirajan ';

let b = "nirajan";

console.log(a + b + "Khatiwada"); // Output: ' My name is nirajan nirajanKhatiwada'

1. **Simple Form (String Boilerplate)**
   * **Explanation:** Uses template literals to embed expressions within a string.
   * **Syntax:**

`${expression}`;

* + **Returns:** A new string with embedded expressions evaluated.
  + **Example:**

let a = ' My name is nirajan ';

let b = "nirajan";

console.log(`${a}${b}khatiwada`); // Output: ' My name is nirajan nirajankhatiwada'

1. **Accessing Element of String**
   * **Explanation:** Retrieves the character at a specified index.
   * **Syntax:**

string[index];

* + **Returns:** The character at the given index (or undefined if out of range).
  + **Example:**

let a = ' My name is nirajan ';

console.log(a[0]); // Output: ' '

1. **Finding Length of String**
   * **Explanation:** Gets the number of characters in the string.
   * **Syntax:**

string.length;

* + **Returns:** The length of the string as a number.
  + **Example:**

let a = ' My name is nirajan ';

console.log(a.length); // Output: 21

1. **To Uppercase**
   * **Explanation:** Converts all characters in the string to uppercase.
   * **Syntax:**

string.toUpperCase();

* + **Returns:** A new string with all characters converted to uppercase.
  + **Example:**

let a = ' My name is nirajan ';

console.log(a.toUpperCase()); // Output: ' MY NAME IS NIRAJAN '

1. **To Lowercase**
   * **Explanation:** Converts all characters in the string to lowercase.
   * **Syntax:**

string.toLowerCase();

* + **Returns:** A new string with all characters converted to lowercase.
  + **Example:**

let a = ' My name is nirajan ';

console.log(a.toLowerCase()); // Output: ' my name is nirajan '

1. **Finding Index of a Substring**
   * **Explanation:** Finds the first occurrence of a specified substring.
   * **Syntax:**

string.indexOf(substring);

* + **Returns:** The index of the first occurrence of the substring (or -1 if not found).
  + **Example:**

let a = ' My name is nirajan ';

console.log(a.indexOf('n')); // Output: 6

1. **String Slicing**
   * **Explanation:** Extracts a section of the string based on start and end indices.
   * **Syntax:**

string.slice(start, end);

* + **Returns:** A new string containing the extracted section.
  + **Example:**

let a = ' My name is nirajan ';

console.log(a.slice(0, 4)); // Output: ' My'

1. **Trim**
   * **Explanation:** Removes whitespace from both ends of the string.
   * **Syntax:**

string.trim();

* + **Returns:** A new string with whitespace removed from both ends.
  + **Example:**

let a = ' My name is nirajan ';

console.log(a.trim()); // Output: 'My name is nirajan'

1. **Replace**
   * **Explanation:** Replaces the first occurrence of a specified substring or pattern with a new substring.
   * **Syntax:**

string.replace(search, replacement);

* + **Returns:** A new string with the specified substring replaced.
  + **Example:**

let a = ' My name is nirajan ';

console.log(a.replace("nirajan", "kirajan")); // Output: ' My name is kirajan '

1. **Split**
   * **Explanation:** Splits the string into an array of substrings based on a specified separator.
   * **Syntax:**

string.split(separator, limit);

* + **Returns:** An array of substrings.
  + **Example:**

let a = ' My name is nirajan ';

console.log(a.split(" ")); // Output: [' My', 'name', 'is', 'nirajan']

### Number Methods and Operations

1. **To Fixed**
   * **Explanation:** Formats a number using fixed-point notation with a specified number of decimal places.
   * **Syntax:**

number.toFixed(digits);

* + **Returns:** A string representing the number with the specified number of decimal places.
  + **Example:**

javascript

Copy code

let c = 10.001;

console.log(c.toFixed(10)); // Output: '10.0010000000'

1. **Math.ceil**
   * **Explanation:** Rounds a number up to the nearest integer.
   * **Syntax:**

javascript

Copy code

Math.ceil(number);

* + **Returns:** The smallest integer greater than or equal to the given number.
  + **Example:**

javascript

Copy code

let a = 1.1000;

console.log(Math.ceil(a)); // Output: 2

1. **Math.floor**
   * **Explanation:** Rounds a number down to the nearest integer.
   * **Syntax:**

javascript

Copy code

Math.floor(number);

* + **Returns:** The largest integer less than or equal to the given number.
  + **Example:**

javascript

Copy code

console.log(Math.floor(a)); // Output: 1

1. **Math.round**
   * **Explanation:** Rounds a number to the nearest integer.
   * **Syntax:**

javascript

Copy code

Math.round(number);

* + **Returns:** The value of the number rounded to the nearest integer.
  + **Example:**

javascript

Copy code

console.log(Math.round(a)); // Output: 1

1. **Math.random**
   * **Explanation:** Returns a pseudo-random floating-point number between 0 (inclusive) and 1 (exclusive).
   * **Syntax:**

javascript

Copy code

Math.random();

* + **Returns:** A floating-point number between 0 (inclusive) and 1 (exclusive).
  + **Example:**

javascript

Copy code

console.log(Math.random()); // Output: A random number between 0 and 1

### ****Array in JavaScript****

* **Definition**: An array is a special variable that can hold more than one value at a time. Each value in the array is called an element, and each element has a numeric index, starting from 0.
* **Example**:

javascript

Copy code

let a = [1, 2, 3, 4]; // An array with four elements

### ****Array Operations****

1. **Indexing in Array**
   * **Accessing Elements**:
     + Arrays are zero-indexed, meaning the first element is at index 0.
     + **Example**:

javascript

Copy code

console.log(a[0]); // Output: 1

console.log(a[3]); // Output: 4

1. **Slicing in Array**
   * **Slicing**:
     + The slice() method extracts a section of the array and returns it as a new array.
     + **Syntax**: array.slice(startIndex, endIndex)
     + **Example**:

javascript

Copy code

console.log(a.slice(0, 2)); // Output: [1, 2] - elements from index 0 up to, but not including, index 2

### ****Array Methods****

1. **Length of Array**
   * **Description**: Returns the number of elements in the array.
   * **Example**:

javascript

Copy code

console.log(a.length); // Output: 4

1. **Push**
   * **Description**: Adds one or more elements to the end of the array.
   * **Example**:

javascript

Copy code

a.push(5);

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

1. **Pop**
   * **Description**: Removes the last element from the array.
   * **Example**:

javascript

Copy code

a.pop();

console.log(a); // Output: [1, 2, 3, 4]

1. **Shift**
   * **Description**: Removes the first element from the array.
   * **Example**:

javascript

Copy code

a.shift();

console.log(a); // Output: [2, 3, 4]

1. **Unshift**
   * **Description**: Adds one or more elements to the beginning of the array.
   * **Example**:

javascript

Copy code

a.unshift(0);

console.log(a); // Output: [0, 1, 2, 3, 4]

1. **Join**
   * **Description**: Joins all elements of an array into a string, separated by a specified separator.
   * **Example**:

javascript

Copy code

let data = a.join(" ");

console.log(data); // Output: "1 2 3 4"

1. **Concatenation of Two Arrays**
   * **Description**: Merges two or more arrays into one.
   * **Example**:

javascript

Copy code

let a2 = [5, 4, 1, 3, 4];

console.log(a.concat(a2)); // Output: [1, 2, 3, 4, 5, 4, 1, 3, 4]

1. **Sort**
   * **Description**: Sorts the elements of an array in place and returns the sorted array.
   * **Example**:

javascript

Copy code

a2.sort();

console.log(a2); // Output: [1, 3, 4, 4, 5]

1. **Reverse**
   * **Description**: Reverses the order of the elements in the array in place.
   * **Example**:

javascript

Copy code

a2.reverse();

console.log(a2); // Output: [5, 4, 4, 3, 1]

1. **Removing Elements from a Specific Position**
   * **Description**: The splice() method changes the contents of an array by removing or replacing existing elements.
   * **Syntax**: array.splice(index, numberOfElementsToRemove)
   * **Example**:

javascript

Copy code

let newData = [1, 2, 3, 4];

newData.splice(1, 2); // Removes 2 elements starting at index 1

console.log(newData); // Output: [1, 4]

1. **Inserting Elements at a Specific Position**
   * **Description**: The splice() method can also be used to insert elements into the array.
   * **Syntax**: array.splice(index, 0, element1, element2, ...)
   * **Example**:

javascript

Copy code

let lasrData = [1, 4];

lasrData.splice(1, 0, 2, 3); // Inserts elements 2 and 3 at index 1

console.log(lasrData); // Output: [1, 2, 3, 4]

1. **Spread Operator (...)**
   * **Description**: The spread operator allows you to spread out elements of an array into another array or function arguments.
   * **Example**:

javascript

Copy code

let finalData = [...newData, ...lasrData];

console.log(finalData); // Output: [1, 4, 1, 2, 3, 4]

1. **Flat**
   * **Description**: The flat() method creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.
   * **Syntax**: array.flat(depth)
     + depth: Optional. Specifies how deep a nested array structure should be flattened. Defaults to 1.
   * **Example**:

javascript

Copy code

let nestedArray = [1, [2, 3], [4, [5, 6]]];

console.log(nestedArray.flat()); // Output: [1, 2, 3, 4, [5, 6]]

console.log(nestedArray.flat(2)); // Output: [1, 2, 3, 4, 5, 6]

### ****In-Depth Explanation of Objects in JavaScript****

#### **1. Defining an Object**

In JavaScript, an object is a collection of properties, where each property is defined as a key-value pair. Here's an example:

javascript

Copy code

const myObject = {

"name": "nirajan", // String property

"class": "bachelor", // Another string property

"is\_topper": "No", // Boolean property (as a string)

"greet": function (from) { // Method (a function inside an object)

console.log(`welcome ${this.name}. from ${from}`);

}

};

* **Keys**: "name", "class", "is\_topper", and "greet" are the keys.
* **Values**: The values associated with these keys are "nirajan", "bachelor", "No", and a function.
* **Method**: "greet" is a method, meaning it's a function stored as a value in the object.

#### **2. Accessing Object Properties**

There are two common ways to access properties in an object:

* **Dot Notation**:

javascript

Copy code

console.log(myObject.name); // Output: nirajan

This is the most common and preferred method when you know the property name in advance.

* **Bracket Notation**:

javascript

Copy code

console.log(myObject['name']); // Output: nirajan

Bracket notation is useful when the property name is stored in a variable or when it contains special characters or spaces.

#### **3. Changing the Value of a Specific Key**

You can easily modify the value of a property in an object by reassigning it:

javascript

Copy code

myObject.name = "kirajan";

console.log(myObject.name); // Output: kirajan

This changes the value associated with the "name" key from "nirajan" to "kirajan".

#### **4. Adding a New Key-Value Pair**

You can add new properties to an existing object by simply assigning a value to a new key:

javascript

Copy code

myObject.lol = "lol";

console.log(myObject.lol); // Output: lol

Now, the myObject object has a new property "lol" with the value "lol".

#### **5. Functions in Objects**

A function defined inside an object is called a **method**. The greet method in myObject is an example:

javascript

Copy code

myObject.greet("sjirajan");

// Output: welcome kirajan. from sjirajan

* **The this Keyword**:
  + The this keyword refers to the object from which the method was called. In the greet method, this.name refers to the name property of myObject. So, when you call myObject.greet("sjirajan"), it accesses myObject.name, which is "kirajan" after the modification.
  + **Example**:

javascript

Copy code

const person = {

name: "John",

greet: function() {

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

}

};

person.greet(); // Output: Hello, my name is John

#### **6. Objects Inside Objects**

Objects can contain other objects, which allows you to structure complex data:

javascript

Copy code

let a = {"detaile": {"name": "nirajan"}};

console.log(a.detaile.name); // Output: nirajan

Here, the a object contains another object as the value of the "detaile" key. You can access nested properties using multiple dot notations.

#### **7. Object Destructuring**

**Destructuring** is a convenient way to extract values from an object and assign them to variables:

* **Basic Destructuring**:

javascript

Copy code

let lol = {"name": "nirajan", "class": 12, "rollno": "11"};

const {name, rollno} = lol;

console.log(name); // Output: nirajan

console.log(rollno); // Output: 11

This creates variables name and rollno that hold the values of the name and rollno properties from the lol object.

* **Renaming During Destructuring**:

javascript

Copy code

const {name: n1, rollno: n2} = lol;

console.log(n1); // Output: nirajan

console.log(n2); // Output: 11

In this example, name is renamed to n1 and rollno to n2.

* **Destructuring in Function Parameters**: You can also use destructuring directly in function parameters to extract specific properties from an object:

javascript

Copy code

function return\_requires({name, rollno}) {

return [name, rollno];

}

console.log(return\_requires({"name": "nirajan", "rollno": 12, "is\_failed": "no"}));

// Output: ["nirajan", 12]

This function extracts the name and rollno properties from the passed object and returns them as an array.

### ****JavaScript Functions****

#### **1. Function Definitions**

JavaScript functions can be defined in several ways:

**Method 1: Function Expression**

const add1 = function(a, b) {

return a + b;

};

* **Description**: A function expression creates a function and assigns it to a variable. This function can be called using the variable name.

**Method 2: Arrow Function**

const add2 = (a, b) => {

return a + b;

};

* **Description**: Arrow functions provide a shorter syntax for writing functions. They also do not have their own this context.

**Method 3: Function Declaration**

function add3(a, b) {

return a + b;

}

* **Description**: A function declaration defines a function with its name. It is hoisted, meaning it can be called before it appears in the code.

**Comparison**: The main difference is that add3,add1 (function declaration) has its own this context, while add1 (function expression) and add2 (arrow function) do not.

**Example Usage**:

console.table([add1(1, 2), add2(2, 3), add3(3, 4)]);

#### **2. Using the Spread Operator**

**Function with Variable Arguments**:

javascript

Copy code

function add(...data) {

let sum = 0;

for (let i = 0; i < data.length; i++) {

sum += data[i];

}

return sum;

}

console.log(add(1, 2, 3)); // Output: 6

* **Description**: The ...data syntax allows the function to accept an indefinite number of arguments as an array.

#### **3. Immediately Invoked Function Expression (IIFE)**

**Example**:

javascript

Copy code

(

function add(a, b) {

console.log(a + b);

}

)(2, 3); // Output: 5

* **Description**: An IIFE is a function that is executed immediately after its definition. It helps in creating a local scope to avoid polluting the global scope.

### Control Flow in JavaScript

**1. Conditional Statements:**

* **if-else Statement:** The if-else statement allows you to execute code blocks based on a condition. The code block inside the if statement is executed if the condition is true. If it's false, the code block inside the else statement is executed.

javascript

Copy code

let a = 2;

if (a === 1) {

console.log(1); // Executes if a is 1

} else if (a === 2) {

console.log(2); // Executes if a is 2

} else {

console.log("None"); // Executes if a is neither 1 nor 2

}

* **switch Statement:** The switch statement evaluates an expression and executes code blocks based on matching case values. Each case represents a value to compare against the expression. The break statement exits the switch block; if omitted, the execution continues into the next case.

javascript

Copy code

switch (a) {

case 1:

console.log(1); // Executes if a is 1

break;

case 2:

console.log(2); // Executes if a is 2

break;

default:

console.log(3); // Executes if a does not match any case

}

**2. Truthy and Falsy Values:**

* **Falsy Values:** These are values that evaluate to false in a boolean context:
  + false – The boolean false value.
  + 0 – The number zero.
  + -0 – The negative zero.
  + 0n – The BigInt zero.
  + "" – An empty string.
  + null – A special value representing the absence of any object value.
  + undefined – A special value representing an uninitialized variable.
  + NaN – Stands for "Not-a-Number," which represents a computation that doesn't yield a number.

javascript

Copy code

console.log(Boolean(false)); // false

console.log(Boolean(0)); // false

console.log(Boolean(-0)); // false

console.log(Boolean(0n)); // false

console.log(Boolean("")); // false

console.log(Boolean(null)); // false

console.log(Boolean(undefined)); // false

console.log(Boolean(NaN)); // false

* **Truthy Values:** Any value that is not falsy is considered truthy:
  + true – The boolean true value.
  + Non-zero numbers – Any number other than 0 (e.g., 1, -1, 3.14).
  + Non-empty strings – Any string that is not empty (e.g., "hello", " ").
  + Objects – Any object, including empty objects {} and arrays [].
  + Functions – Any function definition.
  + Symbols – Values of type Symbol.
  + BigInt – Any BigInt other than 0n (e.g., 1n, -1n).

javascript

Copy code

console.log(Boolean(true)); // true

console.log(Boolean(1)); // true

console.log(Boolean(-1)); // true

console.log(Boolean("hello")); // true

console.log(Boolean(" ")); // true

console.log(Boolean({})); // true

console.log(Boolean([])); // true

console.log(Boolean(function() {})); // true

console.log(Boolean(Symbol())); // true

console.log(Boolean(1n)); // true

**3. Nullish Coalescing Operator (??):**

The nullish coalescing operator (??) is used to provide a default value when dealing with null or undefined. If the left operand is null or undefined, the operator returns the right operand. Otherwise, it returns the left operand.

javascript

Copy code

let val1 = null;

let val2 = val1 ?? 10;

console.log(val2); // Output: 10 (because val1 is null)

**4. Ternary Operator (? :):**

The ternary operator is a shorthand for the if-else statement. It takes three operands: a condition, a result if the condition is true, and a result if the condition is false.

javascript

Copy code

let c = 10;

let b = 10;

let largest = (c > b) ? c : b;

console.log(largest); // Output: 10 (because c and b are equal)

### Loops in JavaScript

**1. For Loop**

javascript

Copy code

// Syntax: for(initialization; condition; increment/decrement) { ... }

for (let i = 0; i < 10; i++) {

console.log(i);

}

**2. While Loop**

javascript

Copy code

// Syntax: while(condition) { ... }

let i = 0;

while (i < 10) {

console.log(i);

i++;

}

**3. Do-While Loop**

javascript

Copy code

// Syntax: do { ... } while(condition);

let i = 0;

do {

console.log(i);

i++;

} while (i < 10);

**4. For-Of Loop**

* **Usage**: Iterates over arrays and strings

javascript

Copy code

const array = [1, 2, 3];

for (const x of array) {

console.log(x);

}

**5. For-In Loop**

* **Usage**: Iterates over the properties of an object, indices of an array, or characters of a string

javascript

Copy code

const obj = {

"name": "nirajan",

"lol": "lol"

};

const arr = ["nirajan", "lol"];

const str = "mynameisnirajan";

for (const key in obj) {

console.log(key); // Prints the keys of the object

}

for (const index in arr) {

console.log(arr[index]); // Prints the values of the array

}

for (const index in str) {

console.log(str[index]); // Prints the characters of the string

}

**6. For-Each Loop**

* **Usage**: Iterates over array elements

javascript

Copy code

const array = [1, 2, 3];

array.forEach((data) => {

console.log(data);

});

**7. Break and Continue Statements**

* **break**: Exits the loop
* **continue**: Skips the current iteration and continues with the next iteration

javascript

Copy code

// Example of break

for (let i = 0; i < 10; i++) {

if (i === 5) {

break; // Exits the loop when i equals 5

}

console.log(i);

}

// Example of continue

for (let i = 0; i < 10; i++) {

if (i % 2 === 0) {

continue; // Skips the rest of the loop body for even numbers

}

console.log(i);

}

### Map, Filter, and Reduce in JavaScript

JavaScript provides powerful array methods such as map, filter, and reduce, which are often used for processing arrays. Here’s a detailed explanation with examples:

**1. filter() Method**

* **Purpose**: Creates a new array with all elements that pass the test implemented by the provided function.
* **Syntax**: array.filter(callback(element[, index[, array]])[, thisArg])

javascript

Copy code

const a = [1, 2, 3, 4, 5, 6];

// Example: Filter out odd numbers

const filtered = a.filter((num) => {

return num % 2;

});

console.log(filtered); // Output: [1, 3, 5]

**2. map() Method**

* **Purpose**: Creates a new array populated with the results of calling a provided function on every element in the calling array.
* **Syntax**: array.map(callback(element[, index[, array]])[, thisArg])

javascript

Copy code

const a = [1, 2, 3, 4, 5, 6];

// Example: Create an array of squares

const squares = a.map((num) => {

return num \* num;

});

console.log(squares); // Output: [1, 4, 9, 16, 25, 36]

**3. Method Chaining**

* **Purpose**: You can chain filter, map, and other array methods together to perform multiple operations in a single statement.

javascript

Copy code

const a = [1, 2, 3, 4, 5, 6];

// Example: Filter out odd numbers and then square them

const result = a.filter((num) => num % 2)

.map((num) => num \* num);

console.log(result); // Output: [1, 9, 25]

**4. reduce() Method**

* **Purpose**: Executes a reducer function (that you provide) on each element of the array, resulting in a single output value.
* **Syntax**: array.reduce(callback(accumulator, currentValue[, index[, array]])[, initialValue])

javascript

Copy code

const a = [1, 2, 3, 4, 5, 6];

// Example: Sum up all the numbers in the array

const sum = a.reduce((accumulator, currentValue) => {

return accumulator + currentValue;

}, 0);

console.log(sum); // Output: 21

**Summary**

* **filter()**: Selects elements that meet a specific condition.
* **map()**: Transforms elements based on a function.
* **reduce()**: Reduces the array to a single value based on a function.
* **Method Chaining**: Combines multiple array operations in a concise and readable manner.

 **in Operator**

* **Objects**: key in obj — Checks if key exists in obj.

javascript

Copy code

"property" in { property: 1 } // true

* **Arrays**: index in arr — Checks if index exists in arr.

javascript

Copy code

0 in [1, 2, 3] // true

 **includes() Method**

* **Strings**: str.includes("substring") — Checks if "substring" is in str.

javascript

Copy code

"hello".includes("he") // true

### Importing and Exporting in JavaScript

#### **Default Export**

* **Only one default export is allowed per module.**
* Use for the primary function, class, or object in a module.

// utils.js

export default function primaryFunction() {

console.log("This is the primary function");

}

javascript

Copy code

// main.js

import primaryFunction from './utils.js';

primaryFunction(); // Output: This is the primary function

#### **Named Export**

* **Allows multiple exports per module.**
* Use to export multiple functions, variables, or objects.

// utils.js

function function1() { console.log("This is function1"); }

function function2() { console.log("This is function2"); }

export { function1, function2 };

javascript

Copy code

// main.js

import { function1, function2 } from './utils.js';

function1(); // Output: This is function1

function2(); // Output: This is function2

### Summary

* **Default Export**: One per module, no curly braces during import.
* **Named Export**: Multiple per module, use curly braces during import.

### Error Handling

**Try-Catch**

The try...catch statement is used for error handling in JavaScript. It allows you to catch exceptions and handle them gracefully without breaking the execution of the program.

javascript

Copy code

try {

// Code that may throw an error

let result = riskyOperation();

} catch (error) {

// Code to handle the error

console.error('An error occurred:', error);

}

### Timers

**setTimeout**

Schedules a function to be executed after a specified delay (in milliseconds).

javascript

Copy code

const timeoutId = setTimeout(() => {

console.log('Executed after 1 second');

}, 1000);

**setInterval**

Repeatedly executes a function at specified intervals (in milliseconds).

javascript

Copy code

const intervalId = setInterval((a, b) => {

console.log(a); // Output: hi

console.log(b); // Output: oi

}, 20, "hi", "oi");

**clearInterval**

Stops a function from being executed repeatedly by clearing the interval.

javascript

Copy code

clearInterval(intervalId);

**clearTimeout**

Cancels a timeout previously established by setTimeout.

javascript

Copy code

clearTimeout(timeoutId);

### Promises

**Promises** represent the eventual completion (or failure) of an asynchronous operation and its resulting value. They are used to handle asynchronous operations in JavaScript.

**Creating a Promise**

javascript

Copy code

const myPromise = new Promise((resolve, reject) => {

if (/\* some condition \*/) {

resolve('Success!');

} else {

reject('Failure!');

}

});

**Using Promises**

javascript

Copy code

myPromise.then(result => {

console.log(result); // Success!

}).catch(error => {

console.error(error); // Failure!

});

### Async/Await

**Async/Await** is syntactic sugar over Promises, making asynchronous code easier to write and read.

**Using Async/Await**

javascript

Copy code

async function fetchData() {

try {

let response = await fetch('https://api.example.com/data');

let data = await response.json();

console.log(data);

} catch (error) {

console.error('Error fetching data:', error);

}

}

### Fetch API

**Fetch** is used to make HTTP requests and returns a promise that resolves to the response of the request.

**Basic Usage**

javascript

Copy code

fetch('https://api.example.com/data')

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

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

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

### Blocking vs Non-Blocking

**Blocking Operations**

Blocking operations are those that halt the execution of subsequent code until they complete. This is common in synchronous operations like reading files or performing complex computations.

**Non-Blocking Operations**

Non-blocking operations, such as those using setTimeout, setInterval, and Promises, allow code execution to continue without waiting for the operation to complete. They typically use asynchronous callbacks or promises.

**Example of Non-Blocking**