JavaScript (ES6)

JavaScript is a versatile programming language primarily used to create dynamic and interactive effects on websites. It is a core technology of the web, alongside **HTML** and **CSS**, and allows web developers to control browser behavior, manipulate web page content, and handle user interactions.

JavaScript **ES6 (ECMAScript 2015)** introduced many new features that significantly improve the language, making it more concise, readable, and powerful.

Let and Const: In ES5, the only way to declare a variable was with var. ES6 introduced let and const for more precise variable scoping.

- let: Declares a block-scoped variable (limited to the block, statement, or expression where it is defined).
- const: Declares a constant (block-scoped) whose value cannot be reassigned.

```
let x = 10; x = 20; // works fine
```

```
const y = 30;

y = 40; // Error: Assignment to constant variable.
```

Arrow Functions: Arrow functions provide a more concise syntax for writing functions and lexically bind the this keyword (i.e., this is inherited from the surrounding scope).

```
// Traditional function

const add = function(a, b) {

  return a + b;

};
```

```
// Arrow function

const addArrow = (a, b) => a + b;

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

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

Template Literals:

Template literals allow embedding expressions inside string literals, using ${} syntax. They also support multi-line strings.

```
const name = "Alice";

const age = 25;

// Using template literals

const message = `Hello, my name is ${name} and I am ${age} years old.`;

console.log(message);  // "Hello, my name is Alice and I am 25 years old."
```

Default Parameters:

ES6 allows functions to have default parameters. If a parameter is not passed, the default value is used.

```
function greet(name = "Guest") {

  console.log(`Hello, ${name}!`);

}

greet();          // "Hello, Guest!"

greet("Alice");    // "Hello, Alice!"
```

Destructuring Assignment:

Destructuring makes it easier to unpack values from arrays or properties from objects into variables.

Array Destructuring:

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


// Destructuring the array

const [a, b] = arr;

console.log(a, b);  // 1 2
```

Object Destructuring:

```
const person = { name: "Alice", age: 25 };


// Destructuring the object

const { name, age } = person;

console.log(name, age);  // Alice 25
```

Spread Operator (...)

The spread operator is used when you want to **expand** or **spread out** elements of an array or object. It is used in situations where you want to unpack or copy values. It's useful for copying arrays/objects and combining them.

Array

```
const arr1 = [1, 2, 3];

const arr2 = [4, 5, 6];

const combined = [...arr1, ...arr2];

console.log(combined);  // [1, 2, 3, 4, 5, 6]
```

Object

```
const obj1 = { a: 1, b: 2 };

const obj2 = { c: 3, d: 4 };

const combinedObj = { ...obj1, ...obj2 };
console.log(combinedObj);

// { a: 1, b: 2, c: 3, d: 4 }
```

Rest Parameter (...)

The rest operator is used to collect or group remaining elements in an array or object into a new array or object. It is often used in function parameters to gather arguments into a single array or object.

In Function Parameters:

```
function sum(...numbers) {

  return numbers.reduce((acc, num) => acc + num, 0);

}

console.log(sum(1, 2, 3, 4)); // 10
```

In Arrays:

```
const [first, ...rest] = [1, 2, 3, 4]

console.log(first)

// 1

console.log(rest)

// [2, 3, 4]
```

In Objects

```
const obj = {a: 1, b: 2, c: 3}

const {a, ...rest} = obj

console.log(a)

// 1

console.log(rest)

// {b: 2, c: 3}
```

Classes:

ES6 introduced the class syntax to create objects and handle inheritance in a more structured and OOP-like manner.

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

Modules (import and export):

ES6 introduced modules to organize and reuse code efficiently. Modules allow you to import and export functionalities between different files, making code more maintainable.

Exporting Modules:

We can export variables, functions, or classes from one file so they can be used in another.

1.Named Export (Multiple Exports)

You can export multiple functions/variables.

math.js (Exporting)

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

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

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

app.js (Importing)

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

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

console.log(subtract(10, 4));
```

- export makes add, subtract, and multiply available for import.
- import { add, subtract } only imports the required functions (not multiply).

2. Default Export (Only One Export)

A module can have one default export.

greet.js (Exporting Default)

```
export default function greet(name) {

  return `Hello, ${name}!`;

}
```

app.js (Importing Default)

```
import greet from "./greet.js";

console.log(greet("John")); // Output:
Hello, John!
```

- export default allows a module to export only one thing.
- The importing file can give any name to the default import (greet in this case).

3. Importing Everything (* as)

If you want to import all exports from a file, use * as.

math.js

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

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

app.js

```
import * as MathOperations from
"./math.js";

console.log(MathOperations.add(5, 3));
// Output: 8

console.log(MathOperations.subtract(1
0, 4)); // Output: 6
```

* as MathOperations imports **everything** from math.js as an object.

We access the functions using MathOperations.add and MathOperations.subtract.

4. Combining Named & Default Exports

A module can have both named exports and a default export.

utils.js

```
export const square = (x) => x * x;

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

export default function sayHello() {

  console.log("Hello, World!");

}
```

app.js

```
import sayHello, { square, cube } from
"./utils.js";

sayHello(); // Output: Hello, World!

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

console.log(cube(2)); // Output: 8
```

- We import the default function (sayHello) without {}.
- We import named exports (square, cube) using {}.

Why Use ES6 Modules?

✅ **Better Code Organization**
✅ **Reusability**
✅ **Efficient Code Splitting**

Type	Export Syntax	Import Syntax
Named Export	export const add = () => {}	import { add } from "./file.js";
Default Export	export default function () {}	import anyName from "./file.js";
Import All	export const x = 5;	import * as Obj from "./file.js";

Promises

A **Promise** in JavaScript is an object that represents the eventual **completion** (or **failure**) of an asynchronous operation and its resulting value.

Why we Use Promises?

JavaScript is **asynchronous**, meaning it doesn't wait for one task to complete before starting another. Promises help handle asynchronous tasks like:
✅ Fetching data from an API
✅ Reading files
✅ Handling timeouts
✅ Avoiding **callback hell**

A **Promise** has three states:

- **Pending** → Initial state (waiting for result)
- **Fulfilled** → Operation completed successfully
- **Rejected** → Operation failed

Creating and Using a Promise

```
const myPromise = new Promise((resolve, reject) => {
  let success = true; // Change to false to see rejection
  setTimeout(() => {
    if (success) {
      resolve("Data fetched successfully!"); // Fulfilled
    } else {
      reject("Error: Data fetch failed!"); // Rejected
    }
  }, 2000);
});
// Handling the Promise
myPromise
  .then((result) => {
    console.log(result); // Runs if resolved
  })
  .catch((error) => {
    console.error(error); // Runs if rejected
  });
```

1. **new Promise((resolve, reject) => { ... })**
   - Creates a Promise that runs an **asynchronous task** (e.g., setTimeout).
2. **Inside setTimeout**
   - If success is true, resolve("Data fetched successfully!") runs (fulfilled).
   - If success is false, reject("Error: Data fetch failed!") runs (rejected).
3. **Handling the Promise:**
   - .then(result => { }) executes if the promise is **fulfilled**.
   - .catch(error => { }) executes if the promise is **rejected**.

Chaining Multiple Promises

We can chain .then() to perform sequential asynchronous tasks.

```
function fetchUser() {
  return new Promise((resolve) => {
    setTimeout(() => resolve({ id: 1, name: "John Doe" }), 1000);
  });
}
function fetchOrders(userId) {
  return new Promise((resolve) => {
    setTimeout(() => resolve(["Order 1", "Order 2", "Order 3"]), 1000);
  });
}
// Chaining Promises
fetchUser()
  .then((user) => {
    console.log(`User: ${user.name}`);
    return fetchOrders(user.id); // Returns another promise
  })
  .then((orders) => {
    console.log("Orders:", orders);
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

1. fetchUser() returns a Promise that resolves with user details.
2. fetchOrders(user.id) returns another Promise with the user's orders.

3. Promise Chaining:
   - First .then() logs the user and returns fetchOrders(user.id).
   - Second .then() logs the orders.

Using Promise.all() (Run Multiple Promises in Parallel)

Use Promise.all() when you want multiple promises to run simultaneously and wait for all of them to complete.

Promise.all takes an array of promises and returns a single promise that resolves when all of the promises in the array have resolved, or rejects if any of the promises reject.

```
const fetchPosts = new Promise((resolve) => {

  setTimeout(() => resolve("Posts Fetched"), 2000);

});

const fetchComments = new Promise((resolve) => {

  setTimeout(() => resolve("Comments Fetched"), 1000);

});

// Execute all promises together

Promise.all([fetchPosts, fetchComments])

  .then((results) => {

    console.log(results); // ["Posts Fetched", "Comments Fetched"]

  })

  .catch((error) => {

    console.error("Error:", error);

  });
```

- Promise.all([fetchPosts, fetchComments]) runs both promises at the same time.
- The .then(results) executes when all promises are fulfilled.
- If any promise fails, .catch() is triggered.

Using Promise.race() (First Promise to Resolve)

Promise.race() returns only the first resolved or rejected promise.

Promise.race takes an array of promises and returns a single promise that resolves or rejects as soon as one of the promises in the array resolves or rejects.

```
const fastAPI = new Promise((resolve) => {
  setTimeout(() => resolve("Fast API Response"), 1000);
});
const slowAPI = new Promise((resolve) => {
  setTimeout(() => resolve("Slow API Response"), 3000);
});
// Returns the first completed promise
Promise.race([fastAPI, slowAPI])
  .then((result) => {
    console.log(result); // "Fast API Response"
  })
  .catch((error) => {
    console.error(error);
  });
```

- Promise.race() waits for the fastest promise to resolve/reject.
- Here, fastAPI finishes first, so its result is logged.

Handling Errors with Promises

Errors can be caught using .catch() in promises or try...catch in async/await.

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => reject("Data fetch failed!"), 2000);
  });
}
// Using .catch()
fetchData()
  .then((data) => console.log(data))
  .catch((error) => console.error("Error:", error));
// Using try...catch in async/await
async function getData() {
  try {
    let data = await fetchData();
    console.log(data);
  } catch (error) {
    console.error("Caught Error:", error);
  }
}
getData();
```

Summary of Promise Methods

Method	Description
new Promise()	Creates a new promise.
.then()	Runs when the promise is fulfilled.
.catch()	Runs when the promise is rejected.
.finally()	Runs after success or failure.
Promise.all([p1, p2])	Runs all promises together; fails if any fail.
Promise.race([p1, p2])	Resolves when the first promise completes.
Promise.allSetelled([p1,p2])	Return all promises together bases the status of the promise.

✅ Promises make asynchronous code **cleaner and structured**.
✅ .then() and .catch() handle success and errors.
✅ Use Promise.all() for parallel execution.
✅ Use Promise.race() to get the **fastest** result.
✅ async/await simplifies promise handling.

Async and Await:

async and await are modern JavaScript features that make working with asynchronous code (like API calls, database queries, or file reading) much easier and more readable. They provide a simpler way to handle Promises without using .then() and .catch().

```
function fetchUserData() {

    return new Promise((resolve, reject) => {

        setTimeout(() => {

            const error = false; // Change to true to simulate an error

            if (!error) {

                resolve({ name: "John Doe", age: 30 });

            } else {

                reject("Failed to fetch user data");

            }

        }, 2000);

    });

}

async function getUser() {

    try {

        console.log("Fetching user data...");

        const user = await fetchUserData();

        console.log("User Data:", user);

    } catch (error) {

        console.log("Error:", error);

    }

}

getUser();
```

When to Use async/await

- When handling **sequential** asynchronous operations.
- When working with **APIs** or **database queries**.

- When you need **cleaner code** instead of .then() chaining.
- When **error handling** with try...catch is required.

✅ async makes a function return a **Promise**.
✅ await pauses the function execution until the **Promise resolves**.
✅ Use try...catch to **handle errors**.
✅ Use Promise.all() for **parallel execution**.

Callback Function:

A **callback function** is a function that is passed as an argument to another function and is executed later, usually after some operation is completed. It allows us to handle asynchronous operations, such as handling API responses, reading files, or executing functions after a specific event.

```
function fetchData(callback) {

  console.log("Fetching data...");

  setTimeout(() => {

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

    callback(data); // Passing fetched data to the callback

  }, 2000);

}

function displayData(data) {

  console.log("Data received:", data);

}

// Calling function with a callback

fetchData(displayData);
```

Callback Hell (also called "Pyramid of Doom") happens when multiple nested callback functions are used, making the code difficult to read and maintain.

```
function fetchUser(userId, callback) {
    setTimeout(() => {
        console.log("User data fetched");
        callback({ id: userId, name: "John Doe" });
    }, 1000);
}
function fetchOrders(user, callback) {
    setTimeout(() => {
        console.log(`Orders fetched for ${user.name}`);
        callback(["order1", "order2", "order3"]);
    }, 1000);
}
function fetchOrderDetails(order, callback) {
    setTimeout(() => {
        console.log(`Details fetched for ${order}`);
        callback({ orderId: order, price: 100 });
    }, 1000);
}
function processPayment(orderDetails, callback) {
    setTimeout(() => {
        console.log(`Payment processed for order ${orderDetails.orderId}, amount:
$${orderDetails.price}`);
        callback("Payment Successful");
    }, 1000);
}
// Callback Hell: Too many nested callbacks
```

```
fetchUser(1, (user) => {
  fetchOrders(user, (orders) => {
    fetchOrderDetails(orders[0], (orderDetails) => {
      processPayment(orderDetails, (status) => {
        console.log("Transaction status:", status);
      });
    });
  });
});
```

Ternary Operator :

The **ternary operator** is a shorthand for if-else statements. It evaluates a condition and returns one of two values based on whether the condition is true or false.

Syntax: condition ? expressionIfTrue : expressionIfFalse;

```
let age = 20
let status = age >= 18 ? "You can vote" : "Minor"
console.log(status)
```

When to use ternary operator:

- **For simple conditional assignments**: When you need to assign a value based on a condition.
- **For inline rendering in JSX (React)**: It keeps code concise.
- **For short and simple conditions**: Helps reduce code length and improves readability.

When NOT to Use the Ternary Operator:

- **When conditions are complex**: Multiple nested ternary operators make the code hard to read.
- **When multiple actions need to be performed**: It’s better to use if-else for better readability.
- **When it reduces readability instead of improving it**: If it makes debugging harder, avoid using it.

```
let score = 85

let grade = score >= 90 ? "A" : score >=
80 ? "B" : score >= 70 ? "C" : "D"

console.log(grade)
```

```
let grade;

if (score >= 90) {

 grade = "A";

} else if (score >= 80) {

  grade = "B";

} else if (score >= 70) {

  grade = "C";

} else {

  grade = "D";

}

console.log(grade);
```

Array Methods:

JavaScript provides a powerful set of **array methods** that help in modifying, searching, iterating, and transforming arrays efficiently.

Mutating (Modifies Original Array):

Method	Description
push()	Adds elements at the end
pop()	Removes the last element
unshift()	Adds elements at the beginning
shift()	Removes the first element
splice()	Adds/removes elements at a specific position
sort()	Sorts an array
reverse()	Reverses an array
fill()	Replaces elements with a static value

Non-Mutating (Returns a New Array)

Method	**Description**
map()	Creates a new array by applying a function
filter()	Returns an array with elements that pass a test
reduce()	Accumulates a value from the array
concat()	Merges two arrays
slice()	Extracts a portion of an array
find()	Finds the first element that matches a condition
findIndex()	Finds the index of the first match
every()	Checks if all elements meet a condition
some()	Checks if at least one element meets a condition
forEach()	Ittirates through each and every eiment of an array and apply the condition won’t create new array

Other Array Methods

Method	Description
includes()	checks if an array contains a certain value, returning true or false.
flat()	creates a new array with sub-array elements concatenated into it up to the specified depth.
flatMap()	maps each element using a mapping function and flattens the result.
reduceRight()	applies a function against an accumulator and each element of the array from right to left.
indexOf()	returns the first index where a value is found, or -1 if it is not found.
join()	joins all elements of an array into a string.
isArray()	Array.isArray() checks if a value is an array.
tostring()	converts an array into a string with comma-separated values.
Array.from()	Array.from() creates an array from an iterable object.

Push: Adds new elements to the end of an array and returns the new length

```
const data = [”apple”, “mango”, “banana”]

data.push(“Grapes”)

console.log(data)
```

Pop: Removes the last element from an array and returns it.

```
const data = [”apple”, “mango”, “banana”]

const data1 = data.pop()

console.log(data1)
```

unshift(): Adds new elements to the beginning of an array.

```
let colors = ["Red", "Green"]

colors.unshift("Blue")

console.log(colors)
```

shift(): Removes the first element and returns it.

```
let queue = ["Person1", "Person2", "Person3"]

queue.shift()

console.log(queue)
```

splice(): Adds or removes elements at a specified index.

syntax:

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

```
let names = ["Alice", "Bob", "Charlie"];

names.splice(1, 1, "David");

console.log(names);
```

sort(): Sorts an array alphabetically (default) or using a custom function.

Syntax : array.sort([compareFunction])

```
let numbers = [4, 2, 9, 1]

numbers.sort((a, b) => a - b)

// Sort ascending

console.log(numbers)
```

reverse(): Reverses the elements in an array.

```
let letters = ["A", "B", "C"]

letters.reverse()

console.log(letters)
```

fill(): Fills array elements with a static value.

```
let arr = new Array(5).fill(0)

console.log(arr)
```

Map: The map() function is a higher-order function in programming that creates a new array or collection by applying a specified function to each element of an existing iterable (like an array or list) without modifying the original iterable.

Purpose: It transforms each element in an array based on the provided function and returns a new array with the transformed elements.

```
array.map(callback(currentValue, index, array))
```

- **callback**: A function that is executed on each element of the array.
- **currentValue**: The current element being processed in the array.
- **index** (optional): The index of the current element.
- **array** (optional): The array map() was called on.

```
let numbers = [1, 2, 3, 4, 5]

let doubledNumbers =
numbers.map(num => num * 2)

console.log(doubledNumbers)
```

```
let people = [

  { name: "John", age: 30 },

  { name: "Jane", age: 25 },

  { name: "Doe", age: 35 }

];

let names = people.map(person =>
person.name);

console.log(names); // ["John", "Jane",
"Doe"]
```

Senarious :
 Converting Data Types : [’1’,’2’,’3’,’4']

 Extracting Specific Values : let product = [{name: ‘apple’, price: 1000},{name:’samsung’, price: 1500}]

Filter : The filter() method in JavaScript creates a new array with all the elements that pass a test.

```
array.filter(callback(element, index, array), thisArg);
```

- **callback**: A function that will be called for every element in the array. It takes three arguments:
  1. **element**: The current element being processed in the array.
  2. **index** (optional): The index of the current element.
  3. **array** (optional): The array that filter() was called on.
- **thisArg** (optional): Value to use as this when executing the callback.

Reducer: The reduce() method is used to iterate over an array and accumulate a single value based on the array's elements. It applies a function to each element of the array (from left to right) and reduces it to a single result (e.g., sum, product, object, etc.).

```
array.reduce(callback(accumulator, currentValue, index, array), initialValue);
```

- **callback**: A function that takes 4 parameters:
  - accumulator: The accumulated value returned from the last iteration.
  - currentValue: The current element being processed in the array.
  - index (optional): The index of the current element.
  - array (optional): The array reduce() was called on.
- **initialValue** (optional): A value to initialize the accumulator. If not provided, the first element of the array will be used.

```
let numbers = [1, 2, 3, 4]

let sum = numbers.reduce((acc, currentValue) => acc + currentValue, 0)

console.log(sum)
```

Scenario : Flattening an Array of Arrays

slice() : The slice() method is used to return a shallow copy of a portion of an array into a new array, without modifying the original array.

```
array.slice(startIndex, endIndex);
```

```
let fruits = ["apple", "banana", "cherry", "date"]

let selectedFruits = fruits.slice(1, 3)

console.log(selectedFruits)
```

Scenario : Removing the First Item in an Array

concat(): The concat() method is used to merge two or more arrays. It returns a new array that contains all the elements from the arrays being concatenated.

```
array.concat(value1, value2, ..., valueN);

let array1 = [1, 2, 3]

let array2 = [4, 5, 6]

let combined = array1.concat(array2)

console.log(combined)
```

find() : The find() method returns the first element in the array that satisfies a provided testing function. If no elements satisfy the condition, it returns undefined.

```
array.find(callback(element, index, array), thisArg);



let numbers = [1, 3, 5, 8, 10]

let firstEven = numbers.find(num => num % 2 === 0)

console.log(firstEven)
```

findIndex() : The findIndex() method returns the index of the first element in the array that satisfies the provided testing function. If no element satisfies the condition, it returns -1.

```
array.findIndex(callback(element, index, array), thisArg);



let numbers = [1, 3, 5, 8, 10]

let index = numbers.findIndex(num => num % 2 === 0)

console.log(index)

// Output: 3
```

every() : The every() method checks if every element in the array passes the test implemented by the provided function. It returns true if all elements satisfy the condition, otherwise false.

```
array.every(callback(element, index, array), thisArg);



let numbers = [1, 2, 3, 4]

let allPositive = numbers.every(num => num > 0)

console.log(allPositive)

// Output: true
```

some() : The some() method checks if at least one element in the array passes the test implemented by the provided function. It returns true if any element satisfies the condition, otherwise false.

```
array.some(callback(element, index, array), thisArg);

let numbers = [1, 2, 3, -4]

let hasNegative = numbers.some(num => num < 0)

console.log(hasNegative)
```

forEach(): executes a provided function once for each array element. The forEach() method is used to execute a function once for each element in an array. It does not return anything but is primarily used for performing side effects such as logging values, updating variables, or modifying elements in the array.

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

numbers.forEach((num) => {

    console.log(num * 2);

});

// Output: 2, 4, 6, 8, 10
```

include(): checks if an array contains a certain value, returning true or false. The includes() method checks whether an array contains a specified value. It returns true if the value exists and false if it does not. This method performs a strict comparison (===), meaning both value and type must match.

```
const fruits = ["apple", "banana", "mango"]

console.log(fruits.includes("banana"))

// true

console.log(fruits.includes("grape"))

// false
```

flat(): creates a new array with sub-array elements concatenated into it up to the specified depth. The flat() method is used to flatten nested arrays. It removes sub-array nesting up to a specified depth, making the array elements more accessible at a single level. The default depth is 1, but a deeper level can be specified.

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

console.log(nestedArr.flat(2));

// Output: [1, 2, 3, 4, 5, 6]
```

flatMap():maps each element using a mapping function and flattens the result. The flatMap() method first maps each element to a new array according to a provided function and then flattens the result by one level. This is useful when you want to transform elements and merge the resulting arrays simultaneously.

```
const numbers = [1, 2, 3];

console.log(numbers.flatMap(num => [num, num * 2]));

// Output: [1, 2, 2, 4, 3, 6]
```

reduceRight(): applies a function against an accumulator and each element of the array from right to left. The reduceRight() method works similarly to reduce(), but it processes elements from right to left instead of left to right. It applies a reducer function that accumulates a value while iterating through the array elements.

```
const words = ["Hello", "World"];

const sentence = words.reduceRight((acc, word) => acc + " " + word);

console.log(sentence);

// Output: "World Hello"
```

indexOf():returns the first index where a value is found, or -1 if it is not found. The indexOf() method returns the first index at which a specified value is found in an array. If the value is not present, it returns -1. This method performs a strict comparison (===).

```
const animals = ["cat", "dog", "elephant"]

console.log(animals.indexOf("dog"))

// 1

console.log(animals.indexOf("lion"))

// -1
```

join(): joins all elements of an array into a string. The join() method converts all elements of an array into a single string, separated by a specified separator. If no separator is provided, it defaults to a comma. This method is commonly used to create readable strings from arrays.

```
const words = ["Hello", "World"];

console.log(words.join(" "));

// Output: "Hello World"
```

tostring(): converts an array into a string with comma-separated values. The toString() method converts an array into a string representation, where elements are separated by commas. This method is similar to join(), but it does not allow specifying a custom separator.

```
const numbers = [1, 2, 3];

console.log(numbers.toString());

// Output: "1,2,3"
```

isArray(): Array.isArray() checks if a value is an array. The Array.isArray() method checks whether a given value is an array. It returns true for arrays and false for any other data type. This is particularly useful when working with functions that accept multiple data types.

```
console.log(Array.isArray([1, 2, 3]))

// true

console.log(Array.isArray("Hello"))

// false
```

Array.from():creates an array from an iterable object. The Array.from() method creates a new array from an iterable object, such as a string, Set, Map, or NodeList. It can also take a mapping function as a second argument to transform the elements while creating the array.

```
const str = "Hello";

console.log(Array.from(str));

// Output: ['H', 'e', 'l', 'l', 'o']



const set = new Set([1, 2, 3]);

console.log(Array.from(set));

// Output: [1, 2, 3]
```