Arrays, Strings and Function

Functions

A function in JavaScript is a block of code that performs a specific task. It takes inputs (parameters) and returns a value or performs an action.

Why Use Functions?

Functions are an essential part of programming in JavaScript. Here are a few reasons why we use functions:

- **Reusability:** You can reuse code by calling the same function multiple times.
- **Modularity:** Functions help in breaking down complex code into smaller, more manageable parts.
- **DRY (Don't Repeat Yourself):** Functions help you avoid writing repetitive code.

Creating a function

Creating a function in JS involves the following steps:

1) Use the `function` keyword followed by the function name.
2) Inside the parentheses, specify any parameters the function will accept (if any).
3) Define the code to be executed by the function inside curly braces.
4) Use the `return` keyword to specify the value to be returned by the function (if any).

Creating a function using parameters:

- Here's an example of creating a function with parameters:

```
function addNumbers(num1, num2) {
  console.log(num1 + num2);
}
```

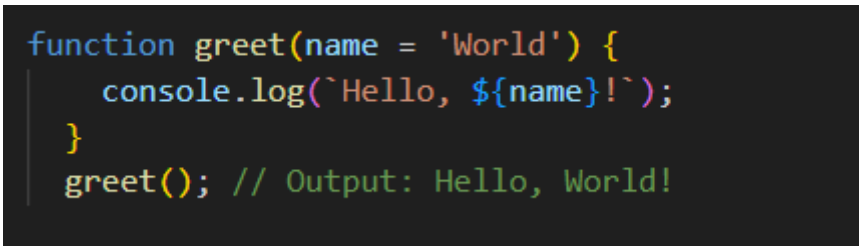
  In this example, `addNumbers` is the function name and `num1` and `num2` are the parameters. The function adds the two parameters and prints the result.

- Consider another example using string literals:
  - String literals can be used in functions in JS by enclosing the string within backticks (` `) instead of single or double quotes. This allows for the use of template literals, which can contain variables and expressions within `${}` that are evaluated and concatenated with the string at runtime.
  - Using string literals in functions can make code more readable and concise by reducing the need for concatenation or escaping characters. It can also make it easier to insert dynamic data into the string output.

```
function greet(message) {
    console.log(`Hello, ${message}!`);
}
greet('Welcome to Coding Ninja');
```

Creating function using Default parameters:

In ES6, we can define default values for function parameters. Default parameters are used when a value is not passed to the function for that parameter. Here's an example:



Parameters vs arguments

Parameters are the variables declared in a function's definition. Arguments are the values passed to the function when it is called.

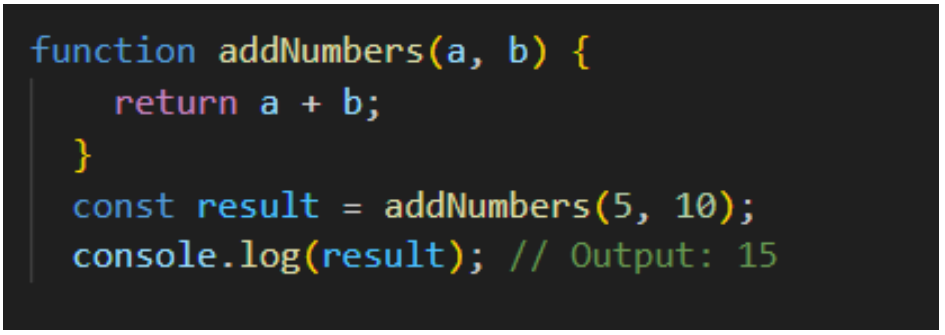
Return statements

- Functions in JavaScript can optionally return a value using the `return` keyword. The value returned by a function can be used in other parts of the code.
- The return statement is used to stop the execution of a function and return a value to the calling code.
- A function can only have a single return statement. This means that once the return statement is executed, the function will immediately terminate and return the value specified by the statement.

- Any statements that appear after the return statement will be considered "unreachable code," meaning they will not be executed. It's important to keep this in mind when designing functions, as any code written after the return statement will not have any effect on the function's output.

Invoking a function

To call a function in JavaScript, we simply need to use the function name followed by parentheses and any arguments (if required) inside the parentheses. Here's an example:



In this example, 5 and 10 act as arguments, and the value of the variable `result` is equal to the function's return value.

Arrays

In JavaScript, an array is a collection of data values of any data type. Arrays are a fundamental data structure and are used to store and manipulate data in many JavaScript programs.

Defining Array in JS:

There are different ways to define an array in JavaScript. They are listed below:

1. The most common way is to use **square brackets** and list out the elements of the array separated by commas. For example:

```
const array_name = [item1, item2, item3];
const myArray = [10, 20, 30, 40, 50];
```

2. Another way to define an array is to use the **`Array() constructor.` For example:**

```
const myArray = new Array(1, 2, 3, 4, 5);
```

3. In JavaScript, we can use the **`Array.from()` method** to create a new array from an existing array-like or iterable object or from a string.

   - `Array.from()` can be used to create an array from a string, where each character in the string becomes an element in the new array, like this:

```
const myString = 'hello';
const newArray = Array.from(myString);
console.log(newArray); // ['h', 'e', 'l', 'l', 'o']
```

   - Arrays can contain any number of elements, separated by commas. The elements in an array can be of any data type, including other arrays, objects, or functions. **For Example:**

```
const myArray = [1, "two", true, ["four", "five"], { six: 6 }];
```

Accessing Array elements:

Arrays in JavaScript are zero-indexed, which means that the first element in an array is accessed using the index 0, the second element is accessed using the index 1, and so on.

1. **Accessing using square brackets:**

```
const myArray = [1, 2, 3, 4, 5];
console.log(myArray[0]); // Output: 1
console.log(myArray[2]); // Output: 3
```

2. **Accessing using `.at()`:**

```
const arr=[1,2,3,4,5];
console.log(arr.at(1));  // o/p: 2
```

Built-in array methods:

JavaScript arrays have several built-in methods that allow you to perform various operations on array elements. Here are some of the most commonly used methods:

1. **`push()`:** adds one or more elements to the end of an array.
    - Note that the `` `push()` `` method in JavaScript returns the new length of the array after adding the element(s) to the end of the array.

```
const state = ["Mumbai", "Delhi", "Kolkata", "Chennai"];
state.push("Lucknow", "Bangalore"); // adds Lucknow and Bangalore in array.
console.log(state.push("Punjab")); // Note: return 7
```

2. **`pop()`:** removes the last element from an array and returns it.
    - Note that the `` `pop()` `` method in JavaScript returns the value of the element that was removed from the end of the array.

```
const state = ["Mumbai", "Delhi", "Kolkata", "Chennai"];
state.pop();
console.log(state); // pops Chennai from the end
console.log(state.pop()); // Note: returns Kolkata
```

3. **`sort()`:** sort the elements of an array in ascending or descending order.

- The `sort()` method sorts the elements of an array as strings in alphabetical order. For example:

```
const state = ["Mumbai", "Delhi", "Kolkata", "Chennai"];
state.sort();
console.log(state); // o/p: [ 'Chennai', 'Delhi', 'Kolkata', 'Mumbai' ]
```

- The `sort()` function sorts numerical values on the basis of their Unicode. For example:

```
const num=[11,32,23,63,57];
num.sort();
console.log(num); // o/p: [ 11, 23, 32, 57, 63 ]
```

4. **`indexOf()`:** returns the index of the first occurrence of a specified element in an array.
   Example:

```
const state = ["Mumbai", "Delhi", "Kolkata", "Chennai"];
let index = state.indexOf("Kolkata");
console.log(index);

// Start from index 3:
const state1 = ["Mumbai", "Delhi", "Kolkata", "Chennai", "Kolkata"];
let index1 = state.indexOf("Kolkata",3);
console.log(index1); // o/p: 4
```

5. **`slice()`:** returns a new array containing a portion of the original array.
   - Negative indices can be used with the `slice()` method to extract elements from the end of an array.
   - Example:

```
const state = ["Mumbai", "Delhi", "Kolkata", "Chennai", "Punjab", "Bangalore", "Rajasthan"];
const slicedState1 = state.slice(2);
const slicedState2 = state.slice(-2); // output: ["Banglore" "Rajasthan"]
console.log("Original: ", state);
console.log("New: ", slicedState1);
console.log("New: ", slicedState2);
```

6. **`splice()`:** update the original array content by removing or adding elements.
    - **Syntax:**

```
array.splice(start, deleteCount, item1, item2,item3);
```

Here,

- `array`: The array to be modified
- `start`: The index at which to start changing the array. If negative, it will begin that many elements from the end of the array.
- `deleteCount`: An integer indicating the number of old array elements to remove. If set to 0, no elements are removed.
- `item1, item2, ... item N`: The elements to add to the array, beginning from the start index. If you don't specify any elements, `splice()` will only remove elements from the array.

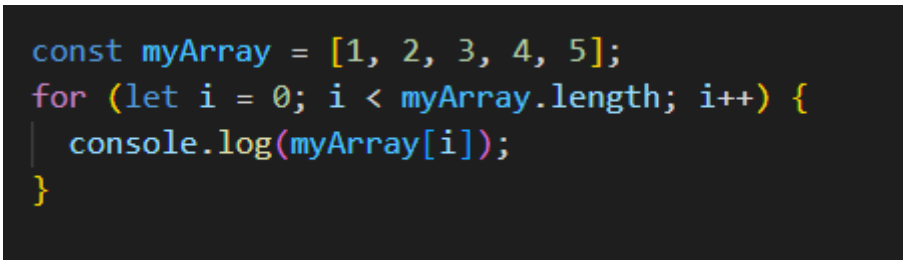
7. **`concat()`:** returns a new array that combines two or more arrays.
   Example:

```
const city1 = ["Jaipur", "Mumbai"];
const city2 = ["Kota", "Shimla", "Gurgaon"];
const mergeCity = city1.concat(city2);
console.log(mergeCity);
```

Iterating over Arrays:

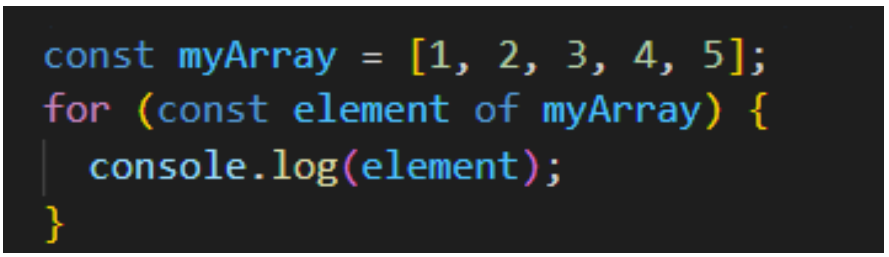
In JavaScript, you can use loops to iterate over the elements of an array. The most common loop types for iterating over arrays are `for` loops, `for...of` loops, and `for...in` loops.

- **Using `for` loop:** The general `for` loop works iterates over an array as shown:



  In this example, the loop iterates over each element of the `myArray` array and logs it to the console.

- **Using `for...of` loop:** A more concise way to iterate over an array is by using this loop shown as below:



  In this example, the `for...of` loop iterates over each element of the `myArray` array and logs it to the console.

  - You can also use other loop types, like `while` and `do...while`, to iterate over arrays, but `for` and `for...of` are the most commonly used.

  - When iterating over arrays using loops, it's essential to remember the array index and length to avoid errors like going out of bounds or skipping elements.

- **Using `for...in` loop:** In JavaScript, the `for...in` loop is used to iterate over the properties of an object (to be covered later), but it can also be used to iterate over the elements of an array. Here's a short example of how to use the `for...in` loop to iterate over an array:

```
let fruits = ['apple', 'banana', 'orange'];

for (let index in fruits) {
  console.log(fruits[index]);
}
```

  In this example, we have an array called `fruits` that contains three elements. We use the `for...in` loop to iterate over the indices of the fruits array, and then use the current index to access the corresponding element of the array using `fruits[index]`.

Note that while it's possible to use the `for...in` loop to iterate over an array, it's generally recommended to use the `for...of` loop instead, as it's specifically designed for iterating over iterable objects like arrays and avoids potential issues with inherited properties and unexpected behavior.

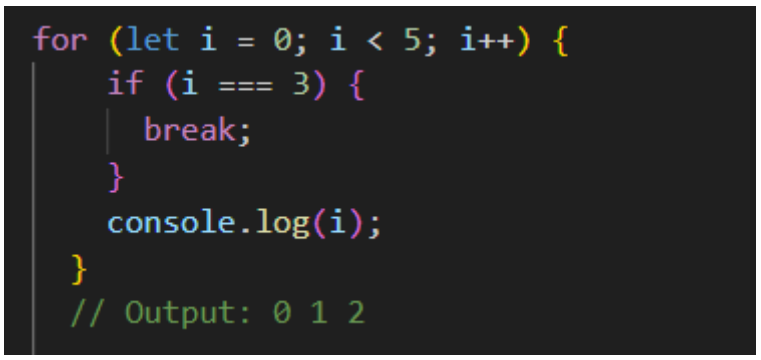
Difference between `for`, `for...of`, and `for...in` loops:

Loop Type	Syntax	Iterates Over	Returns
`for`	for (initialization; condition; iteration) { ... }	Numeric indices of an array or loop counter	Nothing
`for...in`	for (variable in object) { ... }	Property names of an object	Property values
`for...of`	for (variable of iterable) { ... }	Values of an iterable object (e.g. array, string, etc.)	Value

Break and Continue

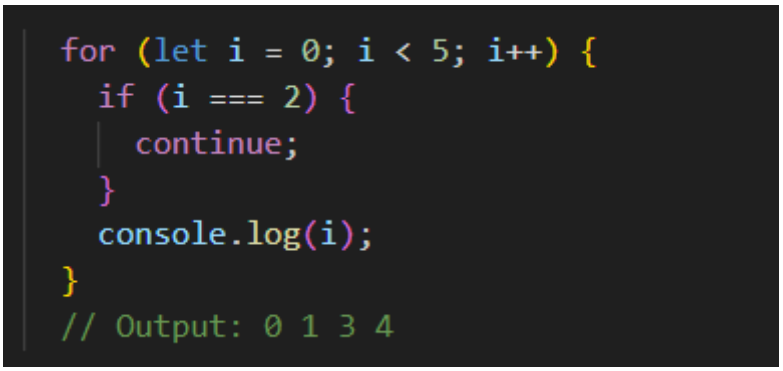
In JavaScript, `break` and `continue` are keywords used inside loops to control the flow of the iteration.

- `break` is used to exit out of a loop entirely when a certain condition is met. Once the `break` statement is executed, the loop will stop and control will move to the next line of code after the loop.
  For example:



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

- `continue` is used to skip the current iteration of a loop and move on to the next iteration. Once the `continue` statement is executed, any code that comes after it within the current iteration will be skipped.
  For example:



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

Spread and Rest Operator

Rest and Spread operators are useful tools in JavaScript that provide flexible ways to work with function arguments and arrays.

1. The `rest` **operator** `...` is used to collect multiple arguments into a single array in a function definition. It allows for a variable number of arguments to be passed to a function and accessed as an array. For example:

```
function sum(...numbers) {
    let total = 0;
    for (let number of numbers) {
      total += number;
    }
    return total;
  }

  console.log(sum(1, 2, 3)); // Output: 6
  console.log(sum(4, 5, 6, 7)); // Output: 22
```

   - In this example, the ...numbers syntax represents the `rest` parameter, which allows us to pass an arbitrary number of arguments to the `sum` function.
   - Inside the function, we use a `for` loop to iterate over the numbers array and add up all the values to get the total sum.
   - The `rest` operator is used here to collect the arguments passed to the function into an array, which we can then work with inside the function.

2. The `spread` **operator** `...` is used to spread the elements of an array into a new array or function arguments. It is commonly used to concatenate arrays or to pass an array of arguments to a function. For example:

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const arr3 = [...arr1, ...arr2];

console.log(arr3); // [1, 2, 3, 4, 5, 6]
```

- In this example, we have two arrays `arr1` and `arr2`. We want to create a new array that contains all the elements from both arrays. We can use the `spread` operator to do this.

- By using `...arr1` and `...arr2`, we are spreading out the elements of the arrays and adding them to the new array `arr3`.
- Finally, we log the new array `arr3` to the console, which contains all the elements from `arr1` and `arr2`.

JavaScript Copying: Shallow vs Deep

- In JavaScript, when using the `spread` operator to copy an object or array, there are two types of copying: shallow and deep.
    - A shallow copy creates a new object or array, but only the top-level values are copied by reference. This means that if any copied values are themselves objects or arrays, they will still be referenced to the original objects or arrays.
    - On the other hand, a deep copy creates a completely new object or array, where all the nested values are also copied by value. This means that all the properties of the original object or array are duplicated, and no references are retained.
- The `spread` operator can be used to create a shallow copy of an object or array but not to create a deep copy. Here's an example of creating a shallow copy using the `spread` operator:

```
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray];

originalArray[0] = 0;

console.log(originalArray); // [0, 2, 3]
console.log(copiedArray); // [1, 2, 3]
```

In this example, we created a new array `copiedArray` using the `spread` operator and assigned the values of `originalArray` to it. When we later modified the first value of `originalArray`, the change was not reflected in `copiedArray`, as it is a separate object.

Objects

Objects are collections of key-value pairs, where each key is a string (also called a "property name"), and each value can be any data type, including other objects. Objects can be used to represent complex data structures, such as arrays or even other objects.

Creating and Accessing Objects

Objects can be created using object literals, constructors, or using the Object.`create()` method. Once an object is created, its properties can be accessed and modified using the dot notation or bracket notation.

Example:

Objects can be created using object literals, enclosed in curly braces {} and using a colon to separate keys from values.

```
const person = {
    name: "John",
    age: 30,
    hobbies: ["reading", "hiking", "cooking"]
  };

  console.log(person.name); // output: John
  console.log(person.hobbies[0]); // output: reading
```

- In the example above, we have an object person with three properties: `name`, `age`, and `hobbies`. The hobbies property is an array of strings.
- To access the properties of the object, we can use dot notation (e.g. `person.name`) or square bracket notation (e.g., `person['name']`).
- To access elements of the hobbies array, we use the square bracket notation with the index of the element we want to access (e.g., `person.hobbies[0]` to access the first element of the hobbies array).

Summarizing it

Let’s summarize what we have learned in this Lecture:

- Different types of Loops (`for`, `while`, and `do while`)
- Function in JS
- Arrays in JS
- Break and Continue
- Spread and Rest operator
- Objects in JS

References

- In-built Array methods: **Link**