Functions in JS

Function Declaration:

- In JavaScript, a function declaration is a way to define a function using the "`function`" keyword followed by the function name and a block of code.
- Function declarations are hoisted, meaning they are moved to the top of the current scope during the compilation phase.
- Having thoroughly covered the fundamental functionality, uses, and features of functions in Lecture-3, we will now delve into an advanced exploration, focusing on different types of functions.
- Understanding the different types of functions in JavaScript empowers developers to choose the appropriate approach for various programming scenarios, leading to cleaner and more efficient code.

In the following section, we will provide a comprehensive discussion on the various types of functions that exist in JavaScript.

Function Expression

- A function expression involves assigning a function to a variable or a property of an object.
- Function expressions are not hoisted and must be defined before they are called.
- They can be anonymous (without a name) or named.
- Function expressions are often used to create functions on the fly, as arguments to other functions, or to encapsulate code within a specific scope.
- Examples of function expressions include anonymous functions, arrow functions, and immediately invoked function expressions (IIFE).

Anonymous Function:

- An anonymous function, also known as a nameless function or function expression, is a function that is defined without a specified name. Instead, it is assigned to a variable or used as an argument directly within the code.
- Anonymous functions are useful when you need to define a small function for a specific task without the need for a named function declaration.

Example:

```
// Using an anonymous function to calculate the square of a
number

const square = function(number) {

  return number * number;

};

console.log(square(5)); // Output: 25
```

Immediately Invoked Function Expression (IIFE):

- An Immediately Invoked Function Expression (IIFE) is a JavaScript design pattern that allows you to execute a function immediately after its declaration. It provides a way to create a private scope for variables and functions, preventing them from polluting the global scope.
- **Syntax:** The basic syntax of an IIFE is as follows:

```
  (function() {

    // Function body

  })();
```

  The function is enclosed in parentheses to indicate that it is a function expression, followed by an additional set of parentheses to invoke it immediately.

- Benefits of using IIFE:

a. Encapsulation: IIFEs create a separate scope for variables and functions, avoiding global scope pollution. This helps prevent naming conflicts and provides a way to encapsulate code and data.
b. Privacy: Variables and functions declared inside an IIFE are not accessible outside of the function, creating privacy and protecting them from external interference.
c. Modularization: IIFEs can be used to create self-contained modules, allowing you to define and expose only specific properties or methods to the outer world.
d. Isolation: IIFEs provide a level of isolation, allowing you to define temporary variables and execute code without affecting the global state or interfering with other parts of the application.

Example:

```
(function() {

  var message = 'Hello, world!';

  function displayMessage() {

    console.log(message);

  }

  displayMessage();

})();
```

Arrow Function:

- An arrow function is a concise syntax introduced in ES6 for defining functions. It offers a more compact and expressive way to write functions, particularly for one-liners.
- It's important to note that arrow functions have a lexically bound "`this`" context, which differs from regular functions. However, since we haven't covered "`this`" in the earlier lectures, we will explore it in detail shortly.

Here are some important points to understand about arrow functions:

1. **Syntax:** The basic syntax of an arrow function is as follows:

```
const functionName = (parameters) => {
  // Function body
};
```

Arrow functions are defined using the => (fat arrow) notation, followed by the function body enclosed in curly braces {}.

2. **Implicit return:** Arrow functions have an implicit return feature. The return statement is not required if the function body consists of a single expression. The result of the expression is automatically returned. For example:

```
const double = (number) => number * 2;

console.log(double(5)); // Output: 10
```

3. **Handling parameters:** Arrow functions can have zero or more parameters. When there is only one parameter, the parentheses around the parameter can be omitted. For multiple parameters, parentheses are required. For example

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

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

const addNumbers = (x, y) => x + y;

console.log(addNumbers(2, 3)); // Output: 5
```

Callback Function

In JavaScript, a callback function is a function that is passed as an argument to another function and is executed later in response to an event or an asynchronous operation. Callback functions provide a way to ensure that certain code is executed only when a specific task is completed or an event occurs.

Here are some important points to understand about callback functions:

1. **Asynchronous programming:** Callback functions are commonly used in asynchronous programming to handle tasks that may take some time to complete, such as reading data from a file, making an HTTP request, or processing a database query.
2. **Event handling:** Callback functions are used to respond to events in web development, such as button clicks, form submissions, or user interactions. The callback function is triggered when the event occurs.
3. **Execution order:** In JavaScript, functions are first-class objects, meaning they can be assigned to variables, passed as arguments, or returned from other functions. A function is not executed immediately when it is passed as a callback. Instead, it is stored and executed later when the conditions are met.

Example:

```
function greet(name, callback) {

  const message = "Hello, " + name + "!";

  callback(message);

}

function displayMessage(message) {

  console.log(message);

}

greet("John", displayMessage); // Output: Hello, John!
```

In this example, the `greet` function takes a `name` and a `callback` function callback as arguments. It constructs a greeting `message` using the `name` parameter and then invokes the callback function with the message.

Pure & Impure Function

Impure Function

- A pure function is a function that always produces the same output for the same inputs and does not cause any side effects.
- It relies only on its arguments and does not modify any external state. Pure functions are predictable, easier to test, and promote code reusability.
- Pure functions are a fundamental concept in functional programming and play a crucial role in building reliable and maintainable code. By following the principles of pure functions, you can minimize side effects, improve code predictability, and make your programs easier to understand and reason about.

Example:

```
// Pure function example
function add(a, b) {
  return a + b;
}
const result = add(3, 4); // Output: 7
```

Impure Function

- An impure function is a function that can produce different outputs for the same inputs or cause side effects by modifying external state.
- It may rely on variables outside its scope or perform actions like modifying global variables or making network requests. Impure functions can be harder to reason about and test.

Example:

```
// Impure function example
let counter = 0;

function incrementCounter() {
  counter++;
  console.log("Counter incremented.");
}
incrementCounter(); // Output: Counter incremented.
```

```
console.log(counter); // Output: 1
```

In this example, the `incrementCounter` function is an impure function that modifies the external variable `counter` and logs a message to the console. Each time `incrementCounter` is called, the `counter` variable is incremented, and the message is logged.

Higher-Order Function

In JavaScript, a higher-order function is a function that can accept other functions as arguments or return a function as its result. Higher-order functions provide a powerful and flexible way to work with functions and enable functional programming paradigms.

Here are some important points to understand about higher-order functions:

- **Function as First-Class Citizens:** In JavaScript, functions are treated as first-class citizens, which means they can be assigned to variables, passed as arguments, and returned from other functions. This allows higher-order functions to be defined and used effectively.
- **Accepting Functions as Arguments:** Higher-order functions can accept other functions as arguments. These functions are often referred to as callback functions or function parameters. The higher-order function can then invoke the callback function at a specific time or condition.
- **Returning Functions:** Higher-order functions can also return functions as their result. This is particularly useful for creating specialized functions or building function factories.
- **Abstraction and Code Reusability:** Higher-order functions promote abstraction and code reusability by encapsulating common functionality in a higher-level function. This can reduce code duplication and make the codebase more modular and maintainable.

Higher Order methods in Array

- Common examples of higher-order functions in JavaScript include **`map()`**, **`filter()`**, **`reduce()`**, and **`forEach()`**, which accept a callback function as an argument. These functions encapsulate common operations on arrays and can be customized by providing different callback functions.

1. `map()`:

- The **`map()`** method is a higher-order function that operates on arrays in JavaScript. It creates a new array by applying a provided callback function to each original array element.
- The callback function is executed for every element, and the return value of each function call is added to the new array.
- The resulting array has the same length as the original array, with each element transformed based on the logic defined in the callback function.

Example:

```
// map() higher-order function example

const names = ["Alice", "Bob", "Charlie", "Dave"];

const nameLengths = names.map(function (name) {

  return name.length;

});

console.log(nameLengths); // Output: [5, 3, 7, 4]
```

In this example, the `map()` function is used to create a new array `nameLengths` that contains the lengths of each name in the `names` array.

2. `filter()`:

- The **`filter()`** method is another higher-order function that works with arrays. It creates a new array containing elements from the original array that satisfy a specified condition.

- The callback function provided to **`filter()`** is executed for each element, and if the return value is true, the element is included in the resulting array. If the return value is false, the element is excluded.

Example:

```
// filter() higher-order function example

const words = ["apple", "banana", "grape", "orange", "kiwi"];

const filteredWords = words.filter(function (word) {

  return word.length > 5;

});

console.log(filteredWords); // Output: ["banana", "orange"]
```

In this example, the `filter()` function is used to create a new array `filteredWords` that contains only the words with a length greater than 5 characters.

3. `reduce()`:

- The **`reduce()`** method is a higher-order function that allows for the accumulation of a single value by iterating over the elements of an array.
- It executes a reducer function on each element and maintains an **accumulator** that stores the accumulated value.
- The reducer function takes two arguments: the **accumulator** and the **current element**. The accumulator is updated based on the logic defined in the reducer function, and the final accumulated value is returned.

Example:

```
// reduce() higher-order function example

const numbers = [1, 2, 3, 4, 5];
const product = numbers.reduce(function (accumulator,
currentValue) {

  return accumulator * currentValue;

}, 1);

console.log(product); // Output: 120
```

In this example, the reduce() function is used to calculate the product of all the numbers in the numbers array.

4. find() and findIndex() functions

The find() and findIndex() methods are array methods introduced in ECMAScript 2015 (ES6) that allow you to search for elements within an array based on a specified condition. Here's an overview of each method and examples of their usage:

find():

- The `find()` method returns the first element in an array that satisfies the provided testing function.
- It executes the callback function once for each element in the array until it finds a match, and then stops the iteration.
- If a matching element is found, it is returned; otherwise, `undefined` is returned.

Example:

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

const foundNumber = numbers.find(function (number) {

  return number > 3;

});

console.log(foundNumber); // Output: 4
```

In this example, the `find()` method is used to search for the first element in the numbers array that is greater than 3. The callback function takes a `number` as an argument and returns `true` if the condition is satisfied. The `find()` method stops iterating as soon as it finds the first match, and returns that element (4 in this case).

findIndex():

- The `findIndex()` method returns the index of the first element in an array that satisfies the provided testing function.
- It executes the callback function once for each element in the array until it finds a match, and then stops the iteration.
- If a matching element is found, its index is returned; otherwise, `` `-1` `` is returned.

Example:

```
const fruits = ["apple", "banana", "grape", "orange"];

const index = fruits.findIndex(function (fruit) {

  return fruit === "grape";

});

console.log(index); // Output: 2
```

In this example, the `findIndex()` method is used to search for the index of the first occurrence of the string "`grape`" in the `fruits` array. The callback function takes a `fruit` as an argument and returns `true` if the condition is satisfied. The `findIndex()` method stops iterating as soon as it finds the first match, and returns the index of that element (`` `2` `` in this case).

Other functions in JavaScript

- There are various functions in-built functions in JavaScript that make it easy to use.
- The array functions given below are powerful tools in JavaScript for manipulating and working with arrays. They provide convenient ways to iterate, search, and modify arrays based on specific conditions or operations.

1. every:

The `every` function tests whether all elements in an array pass a specific condition defined by a provided function. It returns a boolean value indicating if all elements satisfy the condition.

Example:

```
const numbers = [2, 4, 6, 8];

const allEven = numbers.every(function(number) {

  return number % 2 === 0;

});

console.log(allEven); // Outputs: true
```

2. fill:

The `fill` function changes all elements in an array with a static value, starting from a specified start index and ending at a specified end index.

Example:

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

  numbers.fill(0, 2, 4);

  console.log(numbers); // Outputs: [1, 2, 0, 0, 5]
```

3. findLast:

The `findLast` function returns the last element in an array that satisfies a given condition defined by a provided function. It searches the array from right to left.

Example:

```
const numbers = [10, 20, 30, 40, 50];

const foundNumber = numbers.findLast(function(number) {

  return number > 30;

});

console.log(foundNumber); // Outputs: 40
```

4. findLastIndex:

The `findLastIndex` function returns the index of the last element in an array that satisfies a given condition defined by a provided function. It searches the array from right to left.

Example:

```
const numbers = [10, 20, 30, 40, 50];

const foundIndex = numbers.findLastIndex(function(number) {

  return number > 30;

});

console.log(foundIndex); // Outputs: 3
```

5. forEach:

The `forEach` function executes a provided function once for each element in an array. It is commonly used to perform an operation on each item of the array without returning a new array.

Example:

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

numbers.forEach(function(number) {

  console.log(number * 2);

});
```

Function currying

- Function currying is a technique in JavaScript that involves transforming a function with multiple arguments into a sequence of functions, each taking a single argument.
- It allows you to create new functions by partially applying the original function with some arguments, resulting in a more specialized and reusable function.
- Function currying is a powerful technique in JavaScript that enables code reuse, modularity, and composability. It helps create more flexible and specialized functions by partially applying arguments, leading to cleaner and more expressive code.

Example of Function Currying:

```
function multiply(a, b) {

  return a * b;

}

function curriedMultiply(a) {

  return function (b) {

    return multiply(a, b);

  }

}
```

```
const multiplyByTwo = curriedMultiply(2);

console.log(multiplyByTwo(5)); // Output: 10

const multiplyByThree = curriedMultiply(3);

console.log(multiplyByThree(5)); // Output: 15
```

- In this example, the `multiply()` function is a regular function that takes two arguments `a` and `b` and returns their product.
- We define a separate function called `curriedMultiply()` which takes the first argument `a`. Inside this function, we return another function that takes the second argument `b` and calls the `multiply()` function with the provided `a` and `b` arguments.
- This approach achieves function currying by using a separate function (`curriedMultiply()`) to generate the curried functions based on the original `multiply()` function.

this keyword

The `this` keyword in JavaScript refers to the object that is currently executing the code or the object that a function is a method of. It is a special identifier that allows you to access properties and methods within the context of the object.Some of its usage are mentioned below:

1. **Implicit Binding:**

- In most cases, the this keyword is implicitly bound to the object that is left of the dot when invoking a method.
- The value of this is determined dynamically at runtime based on how a function is called.
- It allows methods to access the properties and other methods of the object they belong to.

Example:

```
const person = {
  name: "John",
  age: 25,
  greet: function() {
    console.log(`Hello, my name is ${this.name} and I'm ${this.age}
years old.`);
  }
};
person.greet(); // Output: Hello, my name is John and I'm 25 years
old.
```

In this example, the `greet()` method of the `person` object uses the `this` keyword to access the `name` and `age` properties of the same object. The `this` keyword is implicitly bound to `person` since person is left of the dot when invoking the `greet()` method.

2. **Explicit Binding:**

- In the subsequent lectures, we will delve into the concepts of `call()`, `apply()`, and `bind()`. These powerful functions provide explicit control over the binding of the this keyword to a specific object.
- By covering these functions, you will acquire the knowledge and techniques required to precisely manage the association between `` `this` `` and an object.
- This allows you to control the value of this within a function.

3. **Global Context:**

- In the global scope or when this is not inside any object or function, it refers to the global object (window in browsers, global in Node.js).
- However, in strict mode ("use strict"), the value of this is undefined in the global context.

Example:

```
console.log(this); // Output: Window (in browsers) or Global
```

The behavior of the this keyword can vary depending on how a function is called or the context in which it is used. Understanding the different binding mechanisms of this is essential for properly accessing and manipulating object properties and methods in JavaScript.

Summarizing it

In this set of notes, we covered various concepts related to functions in JavaScript.

- We started with function declaration and function expression, understanding their differences and usage scenarios.
- We explored anonymous functions, which are functions without a name, and saw how they can be used in different contexts
- We then learned about IIFE (Immediately Invoked Function Expression), a technique to create self-contained and private scopes.
- Next, we delved into arrow functions, which provide a concise syntax
- Moving on, we explored callback functions, which are functions passed as arguments to other functions.
- Furthermore, we explored higher-order functions, which are functions that operate on other functions. We covered higher-order functions like `map()`, `reduce()`, `filter()`, `find()`, `findIndex()`, and some other functions which allow for powerful array transformations and operations.
- We also briefly touched on function currying, a technique of creating new functions by partially applying arguments to an existing function and at the end we got our hands on `` `this` `` in JavaScript.

References

- `this` keyword in JavaScript: [Link]()