Hema Coding School

YouTube Link: https://www.youtube.com/@HemaCodingSchool

Arrow Functions

Introduction to Arrow Functions

Arrow functions were introduced in ECMAScript 2015 (ES6) as a shorter syntax for writing function expressions. They provide a more concise way to write functions and come with several differences in behavior compared to traditional function expressions, particularly regarding the handling of the this keyword.

Syntax

The basic syntax of an arrow function is as follows:

```
const functionName = (parameter1, parameter2, ...) => {
   // function body
};
```

Single Parameter, Single Expression:

```
const square = x => x * x;
console.log(square(5)); // Output: 25
```

Multiple Parameters:

```
const add = (a, b) => {
   return a + b;
};
console.log(add(3, 4)); // Output: 7
```

No Parameters:

```

const greet = () => 'Hello, world!';
console.log(greet()); // Output: Hello, world!
```

Returning an Object:

```
const getUser = (name, age) => ({ name: name, age: age });
console.log(getUser('Alice', 25)); // Output: { name: 'Alice', age: 25 }
```

Differences from Traditional Functions

1. **Lexical this:** Arrow functions do not have their own this context. Instead, they inherit this from the enclosing lexical context. This means that this inside an arrow function refers to the this value of the enclosing context.

   Example with Traditional Function:

```
let student = {
    name:"Hema",
    getName:function(){
        return `My name is ${this.name}`
    }
}
let student1 = {
    name:"Hema",
    getName:()=>{
        // return `My name is ${this.name}`
        return `My name is ${student1.name}`
    }
}
console.log(student1.name);
console.log(student1.getName());
```

2. No arguments Object: Arrow functions do not have their own arguments object. If you need to access the arguments object, you should use a traditional function or rest parameters.

```
function objArgument() {
  for (i = 0; i < arguments.length; i++) {
    console.log(arguments[i]);
  }
}
objArgument(1, 2, 3, 4, 5);
-----------------------------------------------------------
let objArgument1 = (...arguments) => {
  for (i = 0; i < arguments.length; i++) {
    console.log(arguments[i]);
  }
};
objArgument1(1, 2, 3, 4, 5);
```

Cannot be used as Constructors: Arrow functions cannot be used as constructors and will throw an error if used with the new keyword.

Example:

```
function Person(name){
    this.name = name;
}
let person1 = new Person("Mahesh")


let Employee = (name)=>{
this.name = name
}
```

```
let employee1 = new Employee("M")
console.log(employee1)
```

Limitations of Arrow Functions

1. **Cannot be used as constructors:** Arrow functions cannot be used with the new keyword.
2. **No arguments object:** They do not have their own arguments object.
3. **No this, super, new.target, and arguments binding:** Arrow functions do not have their own this, super, new.target, and arguments binding. They inherit these from the enclosing lexical context.

Lexical Scope

Definition: Lexical scope, also known as static scope, refers to the scope that is defined at the time of writing code based on the physical structure of the code. In JavaScript, this means that the scope of a variable is determined by its position within the source code and nested functions have access to variables declared in their outer scope.

```
function outerFunction() {
    let outerVar = 'I am an outer variable';
    function innerFunction() {
        console.log(outerVar); // Accesses outerVar from the outer scope
    }
    innerFunction();
}
outerFunction();
```

Closures

Definition: A closure is a feature in JavaScript where an inner function has access to the outer (enclosing) function's variables even after the outer function has returned. Closures allow a function to retain access to its lexical scope, enabling the function to maintain state and data privacy.

```
let outerFunction = () => {
  let a = 12;
  console.log("Hello");
  return innerFunction = () => {
    // console.log(a)
    return a;
  };
};
```

```
let x = outerFunction();
console.log(x());
```

Callbacks

Definition: A callback is a function that is passed as an argument to another function and is executed after some event or operation has occurred. Callbacks are used to handle asynchronous operations, events, and to create more flexible and reusable code.

```
function sayHello(){
   console.log("Hello...")
}
function sayHi(){
   console.log("Hi...")
}
function add(a, b, callBack) {
   callBack();
  console.log(a + b);
}
let a = 2;
let b = 3;
add(a, b, sayHi);
add(a, b, sayHello);
```

Interview Questions:

Arrow Functions

What are arrow functions in JavaScript? How do they differ from regular functions?

Answer: Arrow functions are a concise way to write functions in JavaScript, introduced in ES6. They differ from regular functions in several ways, including lexical this binding, no arguments object, and they cannot be used as constructors.

Explain the concept of lexical this in arrow functions. Provide an example.

Answer: In arrow functions, this is lexically bound, meaning it uses this from the surrounding scope where the arrow function is defined, not where it is called.

```
let student = {
   name:"Hema",
   getName:function(){
      return `My name is ${this.name}`
   }
```

```
}
let student1 = {
   name:"Hema",
   getName:()=>{
      // return `My name is ${this.name}`
      return `My name is ${student1.name}`
   }
}
console.log(student1.name);
console.log(student1.getName());
```

Why can't arrow functions be used as constructors?

Answer: Arrow functions do not have their own this context and do not have a [[Construct]] method, which is required for constructing objects. Using new with an arrow function will throw an error.

What happens if you try to use the arguments object inside an arrow function?

Answer: Arrow functions do not have their own arguments object. If you try to use arguments inside an arrow function, it will refer to the arguments of the enclosing non-arrow function. Use rest parameters (...args) as an alternative.

Lexical Scope

What is lexical scope in JavaScript?

Answer: Lexical scope refers to the scope that is determined at the time of writing code based on the physical structure of the code. Nested functions have access to variables declared in their outer scope.

How does lexical scope differ from dynamic scope?

Answer: Lexical scope is determined by the structure of the code at the time it is written, while dynamic scope is determined at runtime based on the call stack. JavaScript uses lexical scope.

Explain how lexical scope is important for closures in JavaScript.

Answer: Lexical scope allows closures to retain access to variables from their containing scope even after the outer function has returned, enabling powerful patterns like data encapsulation and function factories.

Hema Coding School

YouTube Link: https://www.youtube.com/@HemaCodingSchool

Closures

What is a closure in JavaScript?

Answer: A closure is a function that retains access to its lexical scope, even when the function is executed outside that scope. It allows a function to access variables from its enclosing scope.

Why are closures useful in JavaScript? Provide an example.

Answer: Closures are useful for data encapsulation, creating private variables, and function factories.

How do closures help in creating private variables in JavaScript?

Answer: Closures allow functions to retain access to variables in their enclosing scope, making those variables private to the enclosing function and inaccessible from outside.

Can you provide an example of a closure that maintains state across multiple function calls?

```
function makeAdder(x) {
    return function(y) {
        return x + y;
    };
}
const add5 = makeAdder(5);
console.log(add5(2)); // Output: 7
console.log(add5(10)); // Output: 15
```

Callbacks

What is a callback function in JavaScript?

Answer: A callback function is a function that is passed as an argument to another function and is executed after some event or operation has occurred.

How are callbacks used to handle asynchronous operations in JavaScript?

Answer: Callbacks are used to handle asynchronous operations by executing a function once an operation completes, such as reading a file, making an API call, or setting a timeout.