JavaScript Arrow Function

JavaScript arrow functions are a concise syntax for writing function expressions.

Here's a quick example of the arrow function. You can read the rest of the tutorial for more.

Arrow Function Syntax

The syntax of the arrow function is:

```
let myFunction = (arg1, arg2, ...argN) => {
  statement(s)
}
```

Here,

- myFunction is the name of the function.
- arg1, arg2, ...argN are the function arguments.
- statement(s) is the function body.

If the body has single statement or expression, you can write the arrow function as:

```
let myFunction = (arg1, arg2, ...argN) => expression
```

Example

```
// an arrow function to add two numbers
const addNumbers = (a, b) => a + b;


// call the function with two numbers
const result = addNumbers(5, 3);
console.log(result);


// Output: 8
```

In this example, addNumbers() is an arrow function that takes two parameters, a and b, and returns their sum.

Regular Function vs. Arrow Function

Example 1: Arrow Function With No Argument

If a function doesn't take any argument, then you should use empty parentheses. For example,

```
const sayHello = () => "Hello, World!";
```

```
// call the arrow function and print its return value

console.log(sayHello());


// Output: Hello, World!
```

In this example, when sayHello() is called, it executes the arrow function which returns the string Hello, World!.

Example 2: Arrow Function With One Argument

If a function has only one argument, you can omit the parentheses. For example,

```
const square = x => x * x;



// use the arrow function to square a number

console.log(square(5));


// Output: 25
```

The arrow function square() takes one argument x and returns its square.

Hence, calling square() with the value **5** returns **25**.

---

this Keyword With Arrow Function

Inside a regular function, refers to the function where it is called.

However, this is not associated with arrow functions. So, whenever you call this, it refers to its parent scope. For example,

```
// constructor function

function Person() {


  this.name = 'Jack',

  this.age = 25,

  this.sayAge = function () {


    console.log(this.age);
```

```
        let innerFunc = () => {
            console.log(this.age);
        }


        innerFunc();
    }
}

const x = new Person();
x.sayAge();
```

Output

```
25
25
```

Here, the innerFunc() function is an arrow function.

And inside the arrow function, this refers to the parent's scope, i.e., the scope of the Person() function. Hence, this.age gives **25**.

this Keyword Inside a Regular Function

As stated above, this keyword refers to the function where it is called. For example,

```
// constructor function
function Person() {

    this.name = "Jack",
    this.age = 25,

    this.sayAge = function () {

        // this is accessible
        console.log(this.age);
        function innerFunc() {

            // this refers to the global object
            console.log(this.age);
            console.log(this);

        }

        innerFunc();

    }
}

let x = new Person();
x.sayAge();
```

Output

```
25
undefined
<ref *1> Object [global] {...}
```

Here, this.age inside this.sayAge() is accessible because this.sayAge() is a method of an object.

However, innerFunc() is a normal function and this.age is not accessible because this refers to the global object.

Hence, this.age inside the innerFunc() function is undefined.