AIMERZ.ai
Aim.Act.Achieve

NOTES

# Functions In Javascript - 2

## Introduction to Arrow Functions

Arrow functions were introduced in ES6 (ECMAScript 2015) to simplify function syntax in JavaScript. They provide a shorter way to write functions and behave slightly differently from traditional functions, especially when it comes to handling this context.

**Let us take an example to understand better**

```
JavaScript
// Traditional function
function add(a, b) {
    return a + b;
}

// Arrow function
const add = (a, b) => a + b;

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

## Features of Arrow Functions:

1. **Arrow Syntax (=>):** Arrow functions use a special syntax with the **=> symbol,** which **replaces** the **function keyword** found in **traditional JavaScript functions**. This syntax is **shorter** and more **concise,** making it easier to write functions, especially when they are simple or have minimal code.

**Let us take an example to understand better**

```
JavaScript
// Traditional function
function add(a, b) {
    return a + b;
}

// Arrow function
const add = (a, b) => a + b;
```

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

2. **Implicit Return for Single-Line Expressions:**

If an arrow function has only one expression , we can omit the **return Keyword and curly braces({}).** In the above example , the function implicitly returns the results of the expression. This is useful for writing quick, one-line functions.

**Example with One Parameter:**

```
JavaScript
// Single-line arrow function with implicit return
const square = x => x * x;

console.log(square(4));  // Output: 16
```

In the above example , there's **no return keyword or braces.** The function automatically returns x * x. If there's only one parameter, you can skip the parentheses around it.

**Example with Multiple Parameters:**

```
JavaScript
const multiply = (a, b) => a * b;
console.log(multiply(3, 5));  // Output: 15
```

If there are **multiple statements,** we must use the {} and explicitly use return. Now lets take an example to understand better:

```
JavaScript
const multiplyAndLog = (a, b) => {
    const result = a * b;
    console.log("Result:", result);
    return result;
};

multiplyAndLog(4, 5);  // Output: "Result: 20" and returns 20
```

In the above example, **{} braces** are required, and **return** must be used to return a value.

## Lexical this Binding in Arrow Functions

### What is this ?

In JavaScript, **this** is a special keyword used within functions to refer to the object that is calling the function. However, how **this** behaves can vary based on whether you use a **traditional function** or an **arrow function.**

**Lets understand one by one with example:**

### Traditional Functions: this is Defined by the Calling Object

In **traditional functions,** this is determined by how and where the function is called. So if a function is a method of an object, this refers to that object.

**Example of this in Traditional Functions**

```
JavaScript
const person = {
    name: "Prabir",
    greetTraditional: function() {
        console.log("Hello, my name is " + this.name);
    }
};

person.greetTraditional();  // Output: "Hello, my name is Prabir"
```

In the above example **greetTraditional** is a **traditional function** inside the person object, When we call **person.greetTraditional(),** this refers to person, so this.name is "Prabir".

## Arrow Functions: Lexical this (No Own this Context)

Arrow functions, on the other hand, don't create their own **this** context. Instead, they inherit **this** from the surrounding scope where the function is defined. This means that this in an arrow function is "locked" to the value of **this** in the enclosing (outer) function or scope, rather than being determined by the calling object.

This is called **lexical scoping.**

### Example of Lexical this in Arrow Functions

```
JavaScript
const person = {
    name: "Prabir",
    greetTraditional: function() {
        console.log("Hello, my name is " + this.name);
    },
    greetArrow: () => {
        console.log("Hello, my name is " + this.name);
    }
};

person.greetTraditional();  // Output: "Hello, my name is Prabir"
person.greetArrow();        // Output: "Hello, my name is undefined"
```

**Why Does greetArrow Not Work the Same Way?**

In the above example,**greetTraditional:** Since it's a **traditional function**, **this** is set to person when called with **person.greetTraditional()**, so this.name correctly outputs **"Prabir".**

On the other hand, **greetArrow:** Arrow functions inherit **this** from the surrounding scope at the time the function is defined. In this case, the surrounding scope is the **global scope (or undefined in strict mode)**, where name does not exist. So **this.name is undefined in greetArrow**.

## Arrow Function vs. Traditional Function

| **Feature** | **Traditional Function** | **Arrow Function** |
|---|---|---|
| Syntax | Longer syntax | Shorter, more concise syntax |
| this Context | Own this context | Inherits this from the surrounding scope |
| Usage as Methods | Suitable for methods in objects | Not suitable if this is required in a method |

## Introduction to Recursive Functions

A recursive function is a function that calls itself until it reaches a specific condition to stop. Recursive functions are useful for solving problems that can be broken down into smaller sub-problems.

Before moving forward to example, we will have to know about two terms and its importance -

1. **Base Case:**
   Base case is the condition that determines when the recursive function should stop calling itself. It is the simplest form of the problem that can be directly solved without further recursion.

   It also provides the **termination condition** for the recursive calls, if there will not be any **base case** then it will lead to **infinite recursion**. Also we have to make sure that base case should always be reached at a point else it will lead to infinite recursion.

2. **Recursive Case:**
   It represents the logic where the recursive function calls itself to solve the problem by breaking it down into smaller subproblems. It will be executed again and again until the base case is reached.

**Example-1:**

**In this example,** we are getting a number as input and will have to print all the numbers smaller than that one by one. So if the number will be 3 then the expected output will be 3 2 1 as we are not including the zero in it.

JavaScript
```javascript
function printNumber(number) {
  // base case
  if (number === 0) {
    return;
  }

  console.log(number);
  // recursive case
  return printNumber(number - 1);
}

printNumber(5);

/*
Output
5
4
3
2
1
*/
```

In the above example we can see that we had a **base case** which checks that the number is equal to zero, so that it can return and terminate the recursive call else it will be an infinite recursive call.

Also we are mentioning a **recursive case** which will call the function again by decreasing the value of number by one for the next iteration. This process will continue until the number will be equal to zero and the base case will be reached.

### Example-2

Lets create a **recursive function** which returns the factorial of the given number.
The factorial of a number is the result of multiplying that number by all positive integers smaller than it down to 1. Also the factorial of zero will be 1, which is an edge case to which we have to remember.

JavaScript
```javascript
function factorial(number) {
  // base case
  if (number === 0 || number === 1) {
    return 1;
  }
```

```
    // recursive call
    return number * factorial(number - 1);
  }

  const result = factorial(5);
  console.log(result);

  // Output
  // 120
```

## THANK YOU

AIMERZ.ai

Aim.Act.Achieve

**Stay updated with us!**

Vishwa Mohan

Vishwa Mohan

vishwa.mohan.singh

Vishwa Mohan