Intro to Functions in JavaScript

Defining Functions

Functions are reusable blocks of code that perform a specific task. They allow you to encapsulate logic and execute it whenever needed, making your code more modular and easier to maintain. In JavaScript, functions are first-class citizens, meaning they can be assigned to variables, passed as arguments to other functions, and returned from functions.

Functions in JavaScript are defined using the `function` keyword followed by the function name, parentheses `()`, and curly braces `{}`. Here's an example:

Example 1: Basic Function

```
function sayHello() {
    console.log("Hello!");
}
```

Example 2: Function with Parameters

```
function greet(name) {
    console.log("Hello, " + name + "!");
}
```

Calling Functions

To execute a function, you need to call or invoke it. This is done by using the function name followed by parentheses `()`, optionally passing arguments if the function expects parameters.

```
greet(); // Output: Hello!
```

Function Parameters

Functions can have parameters (input values) and a return statement (output value). Parameters are placeholders for values that will be passed into the function when it is called. Here's an example:

```
function greet(name) {
  console.log("Hello, " + name + "!");
}

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

Return Statement

Functions can also return a value using the `return` statement. The `return` statement specifies the value that the function should return when it completes its execution. The returned value can be used by the caller of the function. Here's an example:

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

const result = add(3, 4);
console.log(result); // Output: 7
```

If a function doesn't have a `return` statement, it will implicitly return `undefined` .

Function Scope

Variables defined inside a function are only accessible within that function. This is known as function scope. Here's an example:

```
function example() {
  const x = 10;
  console.log(x); // Output: 10
}

example();
console.log(x); // Error: x is not defined
```

Pure and Impure Functions in JavaScript

Pure Functions

A **pure function** is a concept derived from functional programming. It refers to functions that have the following characteristics:

1. **Deterministic Output**: The output is determined only by its input values Pure functions will always produce the same output for the same input.
2. **No Side Effects**: The function does not alter any external data (e.g., global variables, I/O operations).

Pure functions are easier to test, debug, and reason about because they depend solely on the input provided and do not involve any hidden data that could affect their output.

```
function multiply(a, b) {
  return a * b;
}

console.log(multiply(3, 4)); // Output: 12
console.log(multiply(3, 4)); // Output: 12
```

This function always returns the same result for the same input, and it does not modify any external state.

Impure Functions

An **impure function** is the opposite of a pure function. It may involve:

1. **Side Effects**: The function might modify some external state, such as changing a global variable, interacting with a database or an API or logging to the console.
2. **Non-deterministic Output**: The output might vary even with the same inputs due to external factors like data stored on disk or random number generation.

Impure functions can be more difficult to manage because their effects and behaviors can depend on the context in which they're executed, making them harder to predict and test.

Example 2: Impure Function with Side Effects

```
let counter = 0;

function incrementCounter() {
    counter += 1;  // Modifies the external 'counter'
variable
    return counter;
}

console.log(incrementCounter());  // Outputs: 1
console.log(incrementCounter());  // Outputs: 2
```

Each call to `incrementCounter` modifies the external state `counter`, making it an impure function.

Example 3: Impure Function with Non-deterministic Output

```
function getCurrentTime() {
    return new Date();  // Output depends on the exact
moment when the function is called
}

console.log(getCurrentTime());  // Output varies
```

The function `getCurrentTime` is impure because it returns a different result with each invocation, despite having no parameters.

Advanced Examples of Pure and Impure Functions Using Arrays in JavaScript

Pure Functions

Example 1: Pure Function to Calculate Sum of an Array

```
function sumArray(numbers) {
    let total = 0; // Initialize total inside the
function scope
    for (let i = 0; i < numbers.length; i++) {
        total += numbers[i]; // Sum up all elements in
the array
    }
    return total; // Return the computed total
}

const myNumbers = [1, 2, 3, 4];
console.log(sumArray(myNumbers)); // Outputs: 10
console.log(sumArray(myNumbers)); // Always outputs: 10,
showing consistency
```

This function is pure because it neither changes the input array nor relies on external state.

Example 2: Pure Function to Merge Two Arrays

```
function mergeArrays(array1, array2) {
    let merged = [];
    for (let i = 0; i < array1.length; i++) {
        merged.push(array1[i]); // Add elements from the
first array
    }
    for (let i = 0; i < array2.length; i++) {
        merged.push(array2[i]); // Add elements from the
second array
    }
    return merged;
}

const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
console.log(mergeArrays(array1, array2)); // Outputs: [1,
2, 3, 4, 5, 6]
```

This function is pure as it returns a new array without modifying the original arrays.

Example 3: Pure Function to Concatenate Two Arrays

```
function concatenateArrays(array1, array2) {
    let result = [];
    for (let i = 0; i < array1.length; i++) {
        result.push(array1[i]);
    }
    for (let i = 0; i < array2.length; i++) {
        result.push(array2[i]);
    }
    return result;
}

const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
console.log(concatenateArrays(array1, array2)); //
Outputs: [1, 2, 3, 4, 5, 6]
```

This function is pure because it does not modify the input arrays and always returns the same output for the same inputs.

Impure Functions

Below are examples involving a simulated "database" represented by a global variable, which can be legitimately modified.

Example 1: Impure Function to Add User to a Simulated Database

```
const userDatabase = [];

function addUserToDatabase(user) {
    userDatabase.push(user); // Modifies the global
'userDatabase'
    return userDatabase;
}

console.log(addUserToDatabase({ name: 'Alice', age: 25
})); // Outputs: [{ name: 'Alice', age: 25 }]
console.log(addUserToDatabase({ name: 'Bob', age: 30 }));
// Outputs: [{ name: 'Alice', age: 25 }, { name: 'Bob',
age: 30 }]
```

This function is impure as it modifies the global state of `userDatabase` .

Example 2: Impure Function to Remove User from Database

```
function removeUserFromDatabase(userName) {
    for (let i = 0; i < userDatabase.length; i++) {
        if (userDatabase[i].name === userName) {
            userDatabase.splice(i, 1); // Modifies the
userDatabase by removing an entry
            break;
        }
    }
    return userDatabase;
}

console.log(removeUserFromDatabase('Alice')); // Outputs:
[{ name: 'Bob', age: 30 }]
```

This function is also impure due to its modification of the global `userDatabase`, affecting the data stored across different parts of the application.

Function Expressions

Function expressions are a way to define functions in JavaScript where the function is assigned to a variable or used as a value. Here are some more examples and explanations:

1. Assigning a function to a variable:

```
const greet = function (name) {
  return "Hello, " + name + "!";
};

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

In this example, we assign an anonymous function to the variable `greet`. The function takes a parameter `name` and returns a greeting string. We can then invoke the function using the variable name followed by parentheses.

2. Using function expressions as arguments:

```
function applyOperation(a, b, operation) {
  return operation(a, b);
}

const add = function (x, y) {
  return x + y;
};

const result1 = applyOperation(3, 4, add);
console.log(result1); // Output: 7

const multiply = function (x, y) {
  return x * y;
};

const result2 = applyOperation(3, 4, multiply);
console.log(result2); // Output: 12
```

Here, we have a function `applyOperation` that takes three parameters: `a`, `b`, and `operation`. The `operation` parameter is expected to be a function. We can pass function expressions as arguments to `applyOperation`, allowing us to define the specific operation to be applied on `a` and `b`.

Function Hoisting

When using function declarations, the entire function is hoisted to the top of its scope, allowing you to call the function before its actual declaration in the code. However, with function expressions, only the variable declaration is hoisted, not the function assignment. Here's an example:

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

function square(x) {
  return x * x;
}

// Function expression
console.log(multiply(3, 4)); // Error: multiply is not a
function

const multiply = function (a, b) {
  return a * b;
};
```

Higher-Order Functions

Higher-order functions are functions that can take other functions as arguments and/or return functions as results. They allow for more abstraction and flexibility in programming. Here's an example:

```
// Higher-order function that applies a transformation to
each element of an array
function mapArray(array, transform) {
  const mappedArray = [];
  for (let i = 0; i < array.length; i++) {
    mappedArray.push(transform(array[i]));
  }
  return mappedArray;
}

// Function to square a number
function square(x) {
  return x * x;
}

// Function to double a number
function double(x) {
  return x * 2;
}

// Using the higher-order function with different
transformation functions
const numbers = [1, 2, 3, 4, 5];

const squaredNumbers = mapArray(numbers, square);
console.log(squaredNumbers); // Output: [1, 4, 9, 16, 25]

const doubledNumbers = mapArray(numbers, double);
console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]

// Using the higher-order function with an inline
function expression
const tripledNumbers = mapArray(numbers, function (x) {
  return x * 3;
});
console.log(tripledNumbers); // Output: [3, 6, 9, 12, 15]
```

In this example, we have a higher-order function called `mapArray` that takes an array and a transformation function as arguments. The `mapArray` function applies the transformation function to each element of the input array and returns a new array with the transformed values.

We define two separate functions, `square` and `double`, which respectively square and double a given number. We then use the `mapArray` higher-order function with these transformation functions to create new arrays:

- `squaredNumbers`: We pass the `numbers` array and the `square` function to `mapArray`, which returns a new array with each number squared.
- `doubledNumbers`: We pass the `numbers` array and the `double` function to `mapArray`, which returns a new array with each number doubled.

We can also use an inline function expression as the transformation function. In the example, we create `tripledNumbers` by passing an inline function that triples each number to `mapArray`.

This example demonstrates how higher-order functions can be used to abstract common operations and make code more reusable. Instead of writing separate loops for squaring, doubling, or tripling the numbers, we can define a generic `mapArray` function that accepts any transformation function. This allows us to apply different transformations to arrays without duplicating code.

Higher-order functions are a powerful concept in functional programming and are widely used in JavaScript for tasks such as mapping, filtering, reducing, and more. They promote code reusability, modularity, and abstraction.