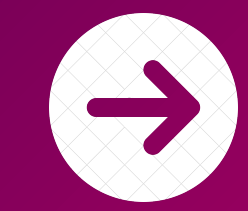
JS Part 1

JS Functional Programming

functional JS ();



WEB OnSet @WebOnset

What is Functional Programming

- JavaScript is a **programming paradigm** use of pure functions, immutability, and first-class functions. It avoids changing state and mutable data.

Key concepts

- **Higher-order functions**
- **Closures**
- **Recursion**
- Popular functional libraries, such as **lodash** and **Ramda,** facilitate functional programming **patterns in JavaScript,**
- This use for **cleaner and more maintainable code.**

What is Paradigm in JS

- Is a **fundamental style** or **approach to programming** that influences how developers **design and structure** their code. It determines the methods and principles used to **solve problems and implement solutions.**

JavaScript supports multiple paradigms

1. **Procedural Programming**
2. **Object-Oriented Programming (OOP)**
3. **Functional Programming**
4. **Event-Driven Programming**
5. **Declarative Programming**

Focus on Functional Programming

*Core Concepts of Functional Programming

1. First-Class Functions

```
// Incorrect use
function operate(x, y) {
    return x + y;
}
console.log(operate(5, 3)); // Output: 8
```

```
// Function assigned to a variable
const add = (a, b) => a + b;

// Function passed as an argument
const operate = (fn, x, y) => fn(x, y);
console.log(operate(add, 5, 3)); // Output: 8
```

2. Immutability

```
let numbers = [1, 2, 3];
numbers.map((num, index, arr) => arr[index] = num * 2);

console.log(numbers); // Output: [2, 4, 6] (Original
array is modified)
```

```
const numbers = [1, 2, 3];
const doubledNumbers = numbers.map(num => num * 2);

console.log(numbers); // Output: [1, 2, 3] (Original
array remains unchanged)
console.log(doubledNumbers); // Output: [2, 4, 6]
```

@Webonset

3. Pure Functions

```
let total = 0;

const addToTotal = (number) => {
    total += number; // Modifies external state
    return total;
}

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

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

console.log(add(2, 3)); // Output: 5
console.log(add(2, 3)); // Output: 5 (Always
returns the same output for the same input)
```

4. Higher-Order Functions

```
function double(number) {
    return number * 2;
}

function applyFunction(fn, num) {
    return fn(num);
}

console.log(applyFunction(5)); // Output:
undefined
```

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

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

5. Function Composition

```
const add = x => x + 1;
const square = x => x * x;

const addThenSquare = x => square(add(x));

console.log(addThenSquare(2)); // Output: 9
(Correct, but less flexible compared to the
`compose` approach)
```

```
const add = x => x + 1;
const square = x => x * x;

const compose = (f, g) => x => f(g(x));
const addThenSquare = compose(square, add);

console.log(addThenSquare(2)); // Output: 9
(i.e., (2 + 1) ^ 2)
```

Simple explanation

- **First-Class Functions:** Functions should be passed around as values.
- **Immutability:** Data should not be modified; new data structures should be created.
- **Pure Functions:** Functions should not produce side effects and should return the same output for the same inputs.
- **Higher-Order Functions:** Functions that take or return other functions.
- **Function Composition:** Combining functions to create new functions.

save post

Webonset

Did you find is Useful!

Follow Us