Intermediate JavaScript Programming

LESSON 4: Functional Programming in JavaScript

Learning Objectives:

By the end of this lesson, participants will be able to:

- Understand the concept of higher-order functions.
- Recognize the benefits and tradeoffs of writing pure functions.
- Apply map, filter, and reduce in chained transformations.
- Write functions that emphasize immutability and reusability.

Lesson Outline:

I. What Is Functional Programming? (10 min)

Functional programming emphasizes the use of functions as building blocks for software. It avoids shared state and side effects, instead relying on transformations of data.

Key Concepts:

- **Functions are first-class**: they can be passed around like values.
- **Pure functions**: always return the same output for the same input, and do not alter global state.
- **Immutability**: avoid changing data; return new versions instead.
- **Higher-order functions**: functions that take other functions as arguments or return them.

Example of a higher-order function:

```
function repeat(operation, n) {
  for (let i = 0; i < n; i++) {
    operation();
  }
}

function sayHi() {
  console.log("Hi");
}

repeat(sayHi, 3);
```

Commentary:

Functional programming isn't all-or-nothing. It's a style you can adopt gradually. The goal is to build small, predictable, reusable pieces.

II. Writing Pure Functions (10 min)

A **pure function** has:

1. No side effects (it doesn't modify variables outside its scope).
2. No reliance on external state.
3. Predictable output for the same input.

Example:

```
function double(x) {
  return x * 2; // Pure function
}

function hypotenuse(a_side, b_side) {
  return Math.sqrt(a_side * a_side + b_side * b_side); // Pure function
}

function addRandom(x) {
  return x + Math.random(); // Not pure: result varies due to randomness,
not external variable state
}
```

Commentary:

Pure functions are easier to test, debug, and reason about. They do not depend on changing external values or produce observable effects.

Note: using stable external constants or libraries (like `Math.PI` or `Math.sqrt`) does not make a function impure. These functions are deterministic and introduce no side effects.

The `hypotenuse` example above is pure, even though it uses an external library function, because the result is entirely predictable and does not modify state. (like `Math.PI` or `Number.isFinite`) does not make a function impure. However, calling a function like `Math.random()`, which produces different results on each call, does — even though it's part of a standard library — because it introduces unpredictability., debug, and reason about. Side effects — such as modifying a global variable, changing a DOM element, or logging — are discouraged inside them.

III. Immutability and Side Effects (10 min)

Mutating shared data leads to hard-to-find bugs. Functional programming encourages **making copies** of data rather than altering the original.

Mutable example:

```
const list = [1, 2, 3];
list.push(4); // modifies original list
```

Immutable approach:

```
const list = [1, 2, 3];
const newList = list.concat(4); // original remains unchanged
```

Commentary:

You don't need to avoid all mutation — but minimize it. Especially avoid mutating data from outside your function.

IV. Method Chaining: map → filter → reduce (20 min)

You can combine functional tools in pipelines:

```
const numbers = [1, 2, 3, 4, 5];
const result = numbers
  .filter(function(n) {
    return n % 2 !== 0; // keep odd
  })
  .map(function(n) {
    return n * n; // square them
  })
  .reduce(function(acc, val) {
    return acc + val; // sum
  }, 0);

console.log(result); // 35
```

Commentary:

Each method returns a new array or value. The original `numbers` array is untouched. This leads to predictable flow. However, overly dense chains can be hard to read — break them up with intermediate variables or helper functions if needed.

V. Building Reusable Functions (5 min)

Instead of writing logic inline, write small functions you can reuse.

```
function isOdd(n) {
  return n % 2 !== 0;
}

function square(n) {
  return n * n;
```

```
}

function sum(acc, val) {
  return acc + val;
}

const result = [1, 2, 3, 4, 5]
  .filter(isOdd)
  .map(square)
  .reduce(sum, 0);

console.log(result); // 35
```

Commentary:

This approach is more verbose, but more readable and easier to test.

VI. Recap & Q&A (5 min)

- Pure functions, immutability, and predictability.
- Higher-order functions and chaining.
- Side effects and code maintenance.

Final Multiple-Choice Question:

Which statement about pure functions is TRUE? A. They rely on global variables for input. B. They always produce different outputs. C. They are easier to test and reason about. D. They modify inputs to save memory.

(Answer: C. They are easier to test and reason about.)