

# Functional programming concepts

Here's a detailed explanation of Functional Programming (FP) concepts in JavaScript:

## ◆ What is Functional Programming?

Functional programming is a **programming paradigm** focused on writing code using **pure functions**, avoiding **shared state**, **mutations**, and **side effects**.

FP treats **computation as the evaluation of mathematical functions**.

## ✓ Core Principles of Functional Programming

Principle	Description
Pure Functions	Functions that do not affect or depend on outside state
Immutability	Data should not be mutated or changed
First-class functions	Functions can be stored in variables, passed as arguments, or returned
Higher-order functions	Functions that take or return other functions
Function composition	Combining multiple functions to create new functionality
Avoid side effects	No changes outside the function scope (e.g., DOM manipulation, API calls)
Declarative code	Describing <i>what</i> to do, not <i>how</i> to do it

## ◆ 1. Pure Functions

| A pure function's output is determined only by its input and has no side effects.

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

// Impure (uses external variable)
let total = 0;
function addToTotal(value) {
  total += value; // Side effect
}
```

## ◆ 2. Immutability

| Data should never be changed. Instead, create and return new copies.

```
const arr = [1, 2, 3];

// Bad (mutation)
arr.push(4);

// Good (immutable)
const newArr = [...arr, 4];
```

Use tools like:

- `Object.assign()`
- Spread operator (`...`)
- Libraries: `Immutable.js`, `Immer`

## ◆ 3. First-Class Functions

| Functions can be assigned to variables, passed as arguments, and returned.

```
const greet = function(name) {  
  return `Hello, ${name}`;  
};  
  
function logGreeting(fn, name) {  
  console.log(fn(name));  
}  
  
logGreeting(greet, 'Abhi');
```

## ◆ 4. Higher-Order Functions

| A function that takes another function as an argument or returns a function.

### Examples:

- `map` , `filter` , `reduce`

```
const numbers = [1, 2, 3, 4];  
  
const doubled = numbers.map(n => n * 2); // [2, 4, 6, 8]  
const evens = numbers.filter(n => n % 2 === 0); // [2, 4]  
const sum = numbers.reduce((acc, val) => acc + val, 0); // 10
```

## ◆ 5. Function Composition

| Combine small functions to build more complex functions.

```
const multiplyBy2 = x => x * 2;  
const square = x => x * x;  
  
const compose = (f, g) => x => f(g(x));
```

```
const squareThenDouble = compose(multiplyBy2, square);
console.log(squareThenDouble(3)); // (3^2) * 2 = 18
```

## ◆ 6. Avoid Side Effects

Side effects include: modifying global variables, writing to disk, DOM manipulation, etc.

Try to isolate side effects or keep them at the edges of your system.

```
// Side effect
function saveUser(user) {
  database.save(user); // avoid unless necessary
}
```

## ◆ 7. Declarative vs Imperative

### Imperative (how to do)

```
const arr = [1, 2, 3];
let result = [];
for (let i = 0; i < arr.length; i++) {
  if (arr[i] % 2 === 0) {
    result.push(arr[i]);
  }
}
```

### Declarative (what to do)

```
const arr = [1, 2, 3];
const result = arr.filter(n => n % 2 === 0);
```

## ◆ 8. Closures

FP relies heavily on closures for building private scopes and partial functions.

```
function multiply(x) {  
  return function(y) {  
    return x * y;  
  };  
}  
  
const double = multiply(2);  
console.log(double(5)); // 10
```

## 🛠 Tools & Libraries Supporting FP

Tool	Use
Ramda	Utility library focused on functional principles
Lodash/fp	FP version of Lodash
Immutable.js	Immutable data structures
RxJS	Functional reactive programming (streams)

## 🧠 When to Use FP in JavaScript

- Complex data transformations
- Stateless services
- Pure UI components (e.g., React Functional Components)
- Avoiding bugs from shared mutable state

## 🚫 Challenges of FP

Challenge	Explanation
Steep learning curve	Requires new mental model

Performance concerns	Too many intermediate objects can be inefficient
Verbosity	Can get verbose without shorthand or chaining

## ✓ Summary

Concept	Description
Pure functions	Same input = same output
Immutability	Avoid changing data
First-class functions	Treat functions like data
Higher-order functions	Accept/return functions
Composition	Combine small functions
Avoid side effects	Minimize external changes

Would you like an example project using functional programming concepts (like a functional Todo app)?