**Functional Programming**

In computer science, functional programming is a programming paradigm where programs are constructed by applying and composing functions.

Functional and object-oriented programming are not opposites. They both will be used alongside.

You need to understand that object-oriented programming is an imperative style of programming, whereas functional programming is a declarative style of programming.

Functional programming is based on mathematical functions. Some of the popular functional programming languages include Python, Erlang, Clojure, etc.

Did you know all of the software problems or use-cases we solve using the object-oriented style of programming can also be solved using Functional programming?

Yes, 100% of all use-cases that can be solved using object-oriented can also be solved using functional programming, the best thing is we have to decide which one suits our use-case when designing the software before writing code.

## 

## What is a Functional Programming?

Functional programming is a programming paradigm designed to handle pure mathematical functions. This paradigm is totally focused on writing more compounded and pure functions.

JavaScript is a multi-paradigm language, which means that we can easily mix a lot of different paradigms inside a simple piece of JavaScript code. We can use object-oriented, procedural and functional programming paradigms all at the same time in JavaScript. The fact that JavaScript is multi-paradigm and allows us to work with a lot of different ways of programming is what makes this language more powerful.

Functional programming has a few important concepts that we need to know and understand. By implementing these concepts in your applications, you will end up with more functional code. This will make a huge difference in your application, making it more readable, usable, manageable, easy to test and bug-free.

Let’s cover the important concepts about functional programming and how we can apply them in JavaScript.

1. Pure Functions.
2. No Side Effects.
3. Immutable data.
4. Recursion
5. First-class functions.
6. Higher-order functions
7. Lazy evaluation etc.

### **First-Class Objects**

Functions in JavaScript are first-class objects, also known as “first-class citizens.” This means that we can work with functions in JavaScript the same way as variables.

const sum = (a, b) => a + b;

const resultSum = sum(1, 2);

const sumAgain = (a, b, sum) => sum(a, b);

JavaScript

Functions in JavaScript are objects—they can have properties and have a link back to their constructor function.

We can work with functions in JavaScript in a lot of different ways. For example:

* We can assign them in a variable as a value.
* We can pass them as arguments to other functions.
* We can return them as a return value of another function.
* We can include them in different data structures.

The way JavaScript handles functions is what makes it possible for us to implement the functional programming paradigm in JavaScript.

### **Pure Functions**

One of the most important concepts in the functional programming paradigm is pure functions. Functional programming requires us to write pure and deterministic functions, and this is what makes functional programming so beautiful and useful: it forces us to write better code with pure functions, making our code easier to test and manage.

Pure functions are functions that, given a specific input, will always return the same output. Pure functions are designed to not cause any side effects—for example, writing to the console, modifying an object, reassigning a variable, etc.

const myName = (name) => `Hello ${name}`;

myName("Leonardo") // Should always return "Hello Leonardo"

JavaScript

One of the reasons that pure functions are so powerful is that they are way easier to test and debug. Since a pure function should return the same output given specific input value, we can test this function very easily.

Some functions rely on data other than the arguments that you pass. These are the impure functions—functions that mutate or change any sort of data, variable, state outside. Impure functions are also known for returning different values.

let count = 0;

const increaseCount = (value) => count += value;

JavaScript

Pure functions are simple and reusable building blocks inside an application. They are completely independent of the outside state, easy to refactor, move around the codebase and adapt in the future.

### **Higher-Order Functions**

A powerful concept of functional programming is higher-order functions. They are a great way to abstract out functionality and write better code.

A high-order function is a function that receives another function as a parameter or returns a function as a return value. You might be using a lot of high-order functions on a daily basis without knowing it—for example, the \_map, filter and reduce\_ methods from JavaScript are higher-order functions.

const names = ["Leonardo", "Lucas", "Bianca", "José", "Maria", "Joe"];

const sayHiToNames = names.map(name => `Hello ${name}`);

JavaScript

It’s very easy to identify if a function is in fact a higher-order function: If this function is receiving another function as a parameter (another term for this is “callback”), it is a higher-order function.

Higher-order functions are very helpful for writing composed and complex functions. They are recommended to abstract and separate your actions, split your functions and create functions that can work with a variety of data types.

### **Composition**

Knowing how to compose your functions and how to put them in the right place is a technique that you can improve over time and a concept that’s very important in functional programming.

Composition can also be expressed as combination—it’s a process of combining multiple functions in a hierarchy to produce a new function or perform a computation.

const splitName = (name) => name.split('\_').join(' ');

const returnNameCapitalized = (name) => name.toUpperCase();

console.log(returnNameCapitalized(splitName('leonardo\_maldonado')));

JavaScript

Composition is all about passing inputs from right to left, returning the output of one function as an input to another function. For beginners in functional programming, this is a more advanced concept and might be hard to grasp at first.

Once you understand and start to use it, you will feel that it helped you to write better composed and more readable functions, separating concerns and making your code cleaner.

### **Declarative vs. Imperative**

Declarative and imperative programming are different programming paradigms that can help you to achieve different results.

Declarative programming is a programming paradigm where we specify the program logic without describing the flow control. Declarative programming is all about **what** to do to achieve a certain result.

const numbers = [1, 2, 3, 4, 5];

const sumNumbers = (n) => n.reduce((acc, current) => acc + current);

JavaScript

Imperative programming is a programming paradigm where we specify the program logic describing the flow control. Imperative programming is all about **how** to achieve a certain result.

const numbers = [1, 2, 3, 4, 5];

const sumNumbers = (n) => {

let finalResult = 0;

for (let i = 0; i < n.length; i++) {

finalResult += n[i];

}

return finalResult;

}

JavaScript

Functional programming favors declarative programming over imperative programming—rather than thinking about **how** to do it, it relies on **what** to do.

### **Immutability**

A very important concept in functional programming is immutability. An immutable object is an object that can’t be modified after its creation. Imagine that we have an object and we want to modify this object by adding a new element—what should we do? We should create a new object, a copy of the actual object, and not mutate it.

Immutability is a powerful concept that helps you to prevent the creation of unexpected side effects in your codebase. It makes it easier for you to read and compose your code.

Since JavaScript is a multi-paradigm language, it does not come with the guarantees that we have in functional programming languages where we can work correctly with the concept of immutability.

To have immutable data in your application, you need to abandon a few methods, especially array methods such as fill, pop, sort, splice, unshift, reverse, push, etc.

We have a few libraries that can help us with that—for example, [Immutable.js](https://immutable-js.github.io/immutable-js/). But sometimes we don’t want to use a third-party library, and that’s the moment that we should rely on vanilla JavaScript for it.

[Object.assign](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign) is the perfect method to work with immutable data. Let’s take as an example an object, and we want to create a new object from this object and modify only one property.

const car = {

model: 'Tesla',

year: 2020

};

const newCar = car;

newCar.model = 'Honda';

JavaScript

We are not using the [Object.assign](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign) method. When we change the newCar object we are also changing the old car object.

Object.assign is an ES6 feature that copies one or more properties from one or more objects to a target object and returns it.

const car = {

model: 'Tesla',

year: 2020

}

const newcar = Object.assign({}, car, {

model: 'Honda'

})

JavaScript

Immutability is a great concept to work with especially when we have big objects and we need to perform some sort of task with them, so we can copy the object and create a new one without mutating it.