Functional programming

Functional programming has become a really hot topic in the JavaScript world. Just a few years ago, few JavaScript programmers even knew what functional programming is, but every large application codebase I’ve seen in the past 3 years makes heavy use of functional programming ideas.

**Functional programming** (often abbreviated FP) is the process of building software by composing **pure functions**, avoiding **shared state,** **mutable data,**and **side-effects**. Functional programming is **declarative** rather than **imperative**, and application state flows through pure functions. Contrast with object oriented programming, where application state is usually shared and colocated with methods in objects.

Functional programming is a **programming paradigm**, meaning that it is a way of thinking about software construction based on some fundamental, defining principles (listed above). Other examples of programming paradigms include object oriented programming and procedural programming.

Functional code tends to be more concise, more predictable, and easier to test than imperative or object oriented code — but if you’re unfamiliar with it and the common patterns associated with it, functional code can also seem a lot more dense, and the related literature can be impenetrable to newcomers.

The hardest part is wrapping your head around all the unfamiliar vocabulary. There are a lot of ideas in the innocent looking definition above which all need to be understood before you can begin to grasp the meaning of functional programming:

* Pure functions
* Function composition
* Avoid shared state
* Avoid mutating state
* Avoid side effects

In other words, if you want to know what functional programming means in practice, you have to start with an understanding of those core concepts.

**Function composition**is the process of combining two or more functions in order to produce a new function or perform some computation. For example, the composition f . g (the dot means “composed with”) is equivalent to f(g(x)) in JavaScript.

**Shared State**

**Shared state** is any variable, object, or memory space that exists in a shared scope, or as the property of an object being passed between scopes. A shared scope can include global scope or closure scopes. Often, in object oriented programming, objects are shared between scopes by adding properties to other objects.

**Immutability**

An **immutable** object is an object that can’t be modified after it’s created. Conversely, a **mutable** object is any object which can be modified after it’s created.

In JavaScript, it’s important not to confuse const, with immutability. const creates a variable name binding which can’t be reassigned after creation. const does not create immutable objects. You can’t change the object that the binding refers to, but you can still change the properties of the object, which means that bindings created with const are mutable, not immutable.

# Side Effects

A side effect is any application state change that is observable outside the called function other than its return value. Side effects include:

1. Modifying any external variable or object property (e.g., a global variable, or a variable in the parent function scope chain)
2. Logging to the console
3. Writing to the screen
4. Writing to a file
5. Writing to the network
6. Triggering any external process
7. Calling any other functions with side-effects

Side effects are mostly avoided in functional programming, which makes the effects of a program much easier to understand, and much easier to test.