ekino.

Functional programming in the JS ecosystem

Version 01 • Février 2020

Contents

Principles of the functional paradigm

01.

functional programming:

" Declarative paradigm of programmation in which functions are first-class citizens. "

OCaml docs

- **Functions** are pure
- **They** are of fixed arity
- **They** do not have any context
- **They** do not produce side effects

Advantages

of functional programming

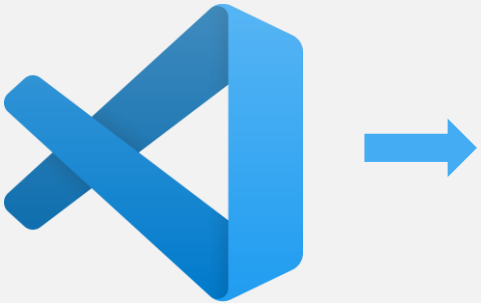
- **No mutations of external context, thus less race conditions**
- **Code describes WHAT the program does, not HOW, thus is easier to refactor and to read**
- **Functions do few tasks (ideally 1), thus are easier to test, and to reuse across the project, or even across different codebases**

Application of FP principles in JS

02.

Currying

i.e. make n unary functions out of one function accepting n arguments.



Partial application

i.e. fixing arguments of a function to produce a new one of smaller arity.

```
const getFormattedLog = (msg, logLevel, time = new Date()) =>
  `${time}  -  ${logLevel}  -  ${msg}`;

const getFormattedError = partial(getFormattedLog, _, "ERROR", _);

getFormattedError("An error has occured");
getFormattedError("An error has occured", new Date("26/05/1993"));
```

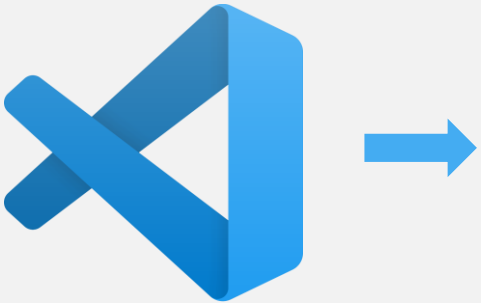
Function composition

i.e. combining multiple functions to create more complicated ones, by flowing the result of each function call to the subsequent.

```
const classAverageMark = testResults =>
    average(getMarks(JSON.parse(testResults)));
```

Memoization

i.e. memoirizing the result of a fonction call to prevent costs of further calls.



Recursive functions

isPrime - loop version

```
const isPrime = (n) => {
    if (n <= 3) return n > 1;
    else if (n % 2 === 0 || n % 3 === 0) return false;

    let i = 5;

    while (i * i <= n) {
        if (n % i === 0 || n % (i + 2) === 0) return false;
        i = i + 6;
    }

    return true;
};
```

Recursive functions

isPrime - recursive version

```
const isPrime = (n, i) => {
    if (n <= 2) return n === 2;
    else if (n % i === 0) return false;
    else if (i * i > n) return true;

    return isPrime(n, i + 1);
};
```

Isolation of side effects

03.

Example:
OpenWeather2DOM

01. **fetch Bordeaux's temp,**
02. **convert it to Celsius,**
03. **multiply by a random number,** ¯_(ツ)_/¯
04. **insert result into the DOM.**

Example:
OpenWeather2DOM

```
const openWeather2DOM = async () => {
    const fahrenheitTemp = await somehowFetchBordeauxTemp();
    const celsiusTemp = (fahrenheitTemp - 32) / 1.8;
    const multipliedByRand = celsiusTemp * Math.random();
    document.querySelector("#temp").innerText = multipliedByRand;
}
```

❌ One single impure function: hard to test, hard to reuse...

Example:
OpenWeather2DOM

```
const fetchBordeauxTemp = async () => fetch('api.openweather.com/bordeaux');

const convertFahrenheitToCelcius = farTemp => (farTemp - 32) / 1.8;

const multiplyByRandom = n => n * Math.random();

const insertIntoDOM = n => (document.querySelector("#temp").innerText = n);
```

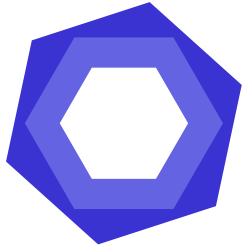
✔ Isolation of side effects / impure functions from pure functions: easier to test, reuse, and mock.

Some helpful tools

04.

TypeScript

Introduces **static typing** as every common functional language have.



ESLint + eslint-plugin-fp

Helps following functional principles by applying a dozen of rules.

RamdaJS or Lodash/FP

Provides many functional programming utility functions like ***curry, partial*** or ***compose***.

ImmerJS or ImmutableJS

Helps maintaining **immutability** by providing natively immutable data structures (for ImmutableJS), or data production helpers (for ImmerJS).

Thanks for listening.
Questions ?

ekino.