

fn(functional
 (programming
 (basics)))

describe(functional-programming)

Functional programming is a way of reasoning with code by expressing computation as the evaluation of functions:

- Functions must be pure
 - No side effects(*)
 - Avoid shared state
- Immutable data structures
- Avoid loops (for, for...each, while, etc...)
- Composition over Inheritance
 - Functions as data
 - Higher order functions

describe(immutability)

Once data has been defined, it should never change (read-only). Changes to data should never mutate the original value. Instead, we should return a *new* copy of the original with any changes being applied to the copy.

```
const obj = { key: "oldValue" };
```

```
Object.assign({}, obj, { key: "newValue" }); ⇒ { key: "newValue" }
```

// or using the spread operator

```
const newObj = { ...obj, foo: "bar" }; ⇒ { key: "oldValue", foo: "bar" }
```

expand(immutability)

Repeatedly copying data structures is quite inefficient. There are several high-performance implementations of immutable data structures in Javascript (the *how* behind persistent immutable data structures is super interesting, albeit, entirely out of scope for this discussion).

Some examples include:

- Immutable.js
- Mori
- Clojurescript

describe(pure-functions)

A pure function should only rely on input passed as arguments.

// consider the following:

```
const identity = x => x;
```

Given the same argument(s), a pure function will always return the same value.

In effect, `identity("foo")` can be replaced with the value `"foo"` & vice versa.

This concept is called *Referential Transparency*

expand(pure-functions)

A pure function should avoid side effects whenever possible.

```
const state = {  
  someValue: "value",  
}; // console.log(state) → { someValue: "foo" }
```

```
const badFn = newValue ⇒ {  
  state.someValue = newValue;  
  return state;  
}; // badFn("foo") → { someValue: "foo" }
```

expand(pure-functions)

```
const state = {  
  someValue: "value",  
};  
  
// console.log(state) → { someValue: "value" }
```

```
const assoc = (obj, k, v) ⇒ {  
  const updatedObj = { ...obj };  
  return (updatedObj[k] = v);  
};  
  
// assoc(state, "someValue", "updated") → { someValue: "updated" }
```

describe(recursion)

Recursion is one of the most important tools in the functional programmer's arsenal.

Proper recursion allows programmers to maintain referential transparency while performing iterative-like operations without variable reassignment or internal state.

expand(recursion)

// consider the following:

An iterative function that calculates the factorial of **n**

```
const iterativeFactorial = n => {  
  let result = 1;  
  for (let i = 1; i ≤ n; i++) {  
    result *= i;  
  }  
  return result;  
}
```

expand(recursion)

Both the variables **result** & **i** are repeatedly being reassigned in this procedural approach.

We want to avoid these sorts of operations if at all possible. Let's write a recursive function to calculate the factorial value of **n**.

expand(recursion)

// calculate n factorial recursively

```
const recursiveFactorial = (n, product = 1) => {  
  if (n === 0) {  
    return product;  
  }  
  return recursiveFactorial(n - 1, product * n);  
}
```

// factorial(3, 1) => factorial(2, 3) => factorial(1, 6) => factorial(0, 6) => 6

expand(recursion)

// flatten a nested array

```
const flatten = (n) => {  
  if (!Array.isArray(n)) {  
    return n;  
  } else {  
    return n.reduce((acc, x) => {  
      return acc.concat(flatten(x));  
    }, []);  
  }  
}
```

// flatten([1, [2], [3, [4]]]) → [1, 2, 3, 4]

describe(functional-composition)

How do we take what we've *just* learned and put it into practice? With higher order functions.

Higher order functions are simply functions that receive functions as arguments and/or return them.

```
const func = (fn) => fn();
```

```
// (remember, because of referential transparency,  
// functions can be viewed as expressions representing  
// data)
```

expand(functional-composition)

Currying (named after Haskell Curry, not the food) is a functional technique where a multi-arity function is expressed as a sequence of functions with a single argument.

```
// normal
```

```
const add = (x, y) => x + y;
```

```
// curried
```

```
const add = x => y => x + y;
```

expand(functional-composition)

This allows us to partially apply arguments to functions
& assign them to variables

```
const add = x ⇒ y ⇒ x + y;
```

```
const inc = add(1); ⇒ const inc = (1, y) ⇒ 1 + y;
```

```
inc(2); ⇒ 3
```

expand(functional-composition)

//consider the following

// how could we parse ["oof", ["rab", ["zab"]]] into a single array of upper-cased, reversed strings?

```
const pipe = (... fns) => (val) => fns.reduce((v, fn) => fn(v), val);
```

```
const reverse = (string) => string.split("").reverse().join("");
```

```
const upperCase = (string) => string.toUpperCase();
```

```
const upperCaseReverse = pipe(reverse, upperCase);
```

```
const processStringArr = (arr) => arr.map(upperCaseReverse);
```

```
const parseData = pipe(flatten, processStringArr);
```


expand(functional-composition)

```
const stringArr = ["oof", ["rab", ["zab"]]];
```

```
parseData(stringArr) ⇒ ["foo", "bar", "baz"]
```

```
console.log(stringArr) ⇒ ["oof", ["rab", ["zab"]]]
```

```
thank(  
  return "Thanks For Coming!"  
);
```

Slides & code are available @ ["https://github.com/gregsugiyama/js-fp"](https://github.com/gregsugiyama/js-fp)