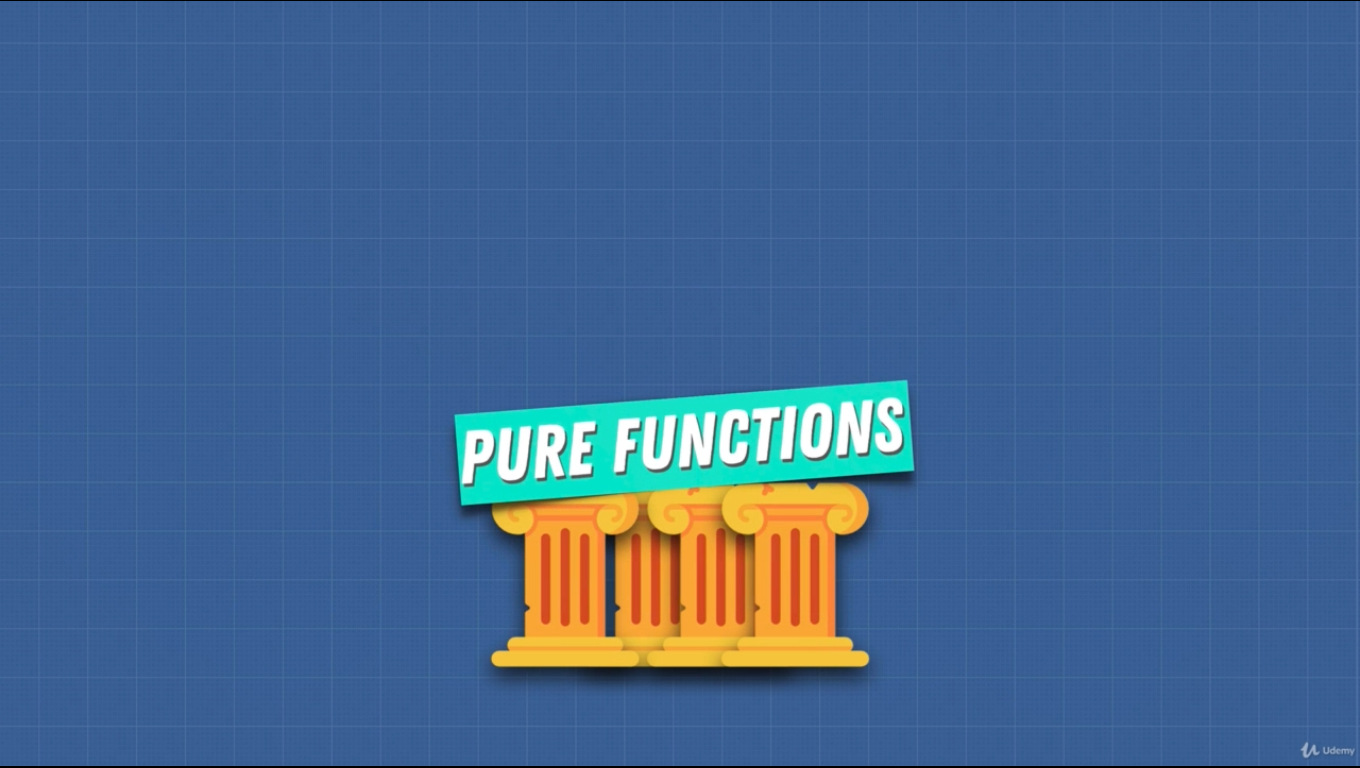
2. Functional Programming Introduction

* In functional programming we don’t combine data and function together.
* Data and function is separated.
* Most functional language don’t have this class and method that belongs to an object.
* Functions operate on data structures (array, object) but don’t belong to them

Goal of functional programming:



Pillar of FP:



* In FP all object that are created are immutable.
* We avoid shared state, we adhere to the principle of pure function.
* FP has a lot of restriction.

4. Pure Functions

1. Function has to always return the same output given the same input.
2. Function can’t modify anything out side of itself. No side effect.

* Order of the function call matters.

5. Pure Functions 2

* console.log modifies the outside world. It’s logging something to the output of the browser. (didn’t get it)

Referential transparency:

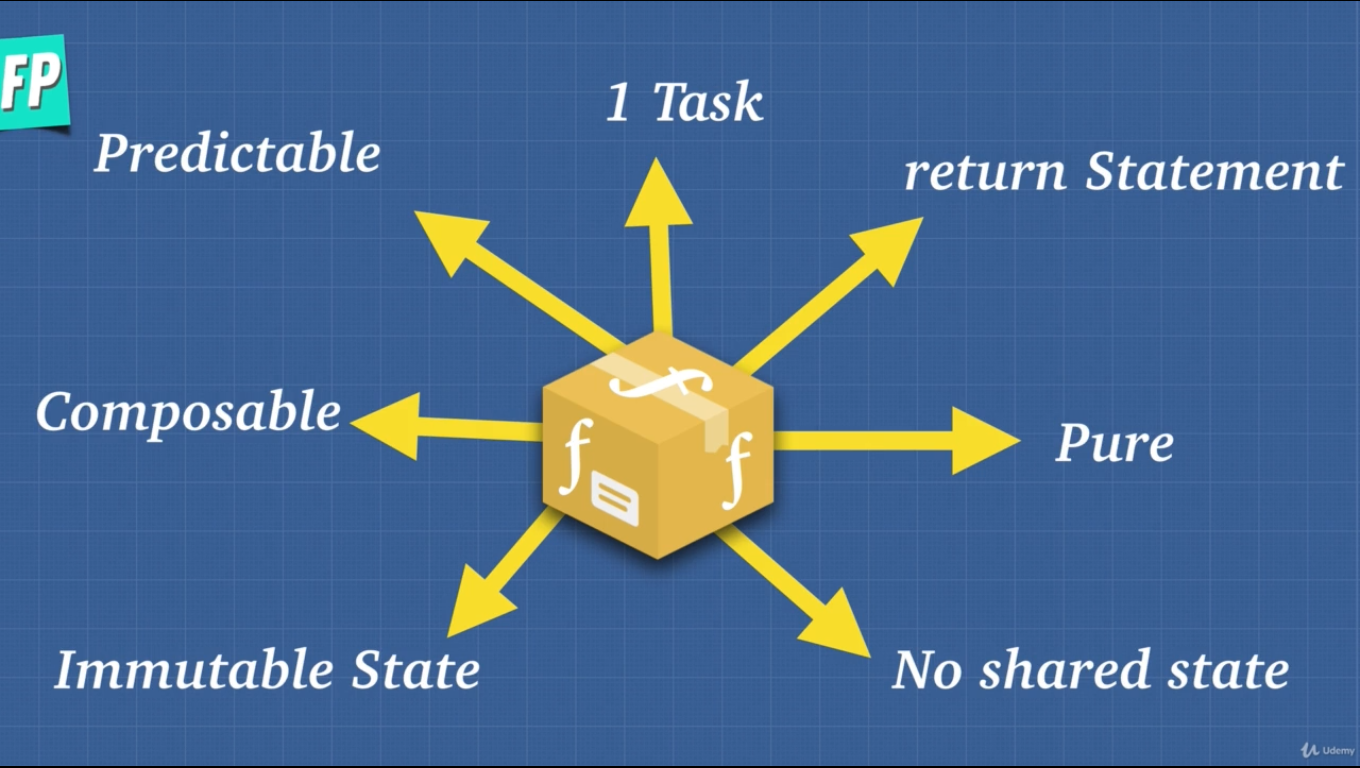
* If I change the function will it then have any effect on the program. It’s kind of testing, I guess.
* Parameters are local variables.

Idea of PF:

* Easy to test.
* Easy to compose.
* Avoids bugs.
  + No mutation .
  + No shared state.

6. Can Everything Be Pure

* console.log side effect
* Input, output are side effects. They are communicating with outside world.
* A program can’t exist without side effect.
* The goal of FP is to minimize side effect.



No shared State:

* With other function.

Immutable State:

* We mutate data inside of a function but we take input and always return a copy of that as output.
* At the end of the day FP is basically making our code predictable.

7. Idempotent

The idea of idempotent is a function does or returns what we expect it to do.

* It’s a bit different than PF, like console.log can be idempotent.
* Deleting a user from DB. We can only delete a user once.
* API calls. Even though we are communicating with the outside world.

8. Imperative vs Declarative

Imperative code that tells the machine what to do and how to do it.

Declarative code that tells the machine what to do and what should happen.

* Computers are better being imperative.
* Humans are declarative.
* Higher level language is more declarative and lower level language more imperative.
* For loop imperative.
* jQuery imperative.
* Modern framework is declarative.
* FP helps us be more declarative.
* Declarative language at the end of the day does imperative work deep down for machine language.
* We make code easy to understand by making it more declarative.

9. Immutability

* The idea in OOP was we could change the name, weapon property.
* In FP the idea of immutability is not changing the state instead making copies of the state and returning a new state every time.
* This idea can be memory inefficient. For this we can use “Structural Sharing”. We only copy the changed part of an object other part remains the same.

10. Higher Order Functions and Closures

HOF: A function that either takes one or more function as arguments or returns a function as a result often called a callback.

* In closure a function remember a variable from the outer scope. It will be available even after the outer function has stopped running.
* Closure only makes a function impure only if we modify the closed over variable (outer variable).
* Using closure, we have data privacy.
* Closure is used very much in FP for data privacy.

11. Currying

Currying: Function takes one parameter at a time.

* This is useful for creating multiple utility function.

12. Partial Application

Partial Application: Process of producing a function with a smaller number of parameters.

* On the second call it expects all the arguments.
* Currying takes one argument at a time.

14. MCI Memoization 1

* For good performance.
* To understand dynamic programming, we need to understand what caching is.
* Memoization is a specific form of caching.
  + Caching the return value of a function based on its parameters.
  + If the parameter remains the same then it is memoized.
  + It calculates only the first time around on the second time it simply returns the cached value.
* It is a way to remember a solution to a solved problem.

15. MCI Memoization 2

* It’s a good practice to have memory or cache inside of a function. Not polluting the global scope.
* To improve the previous example, we simple use closure which is storing the outer function value. BTW we take the global object inside of the function and the logic is put another function.
* It optimize our code.

16. Compose and Pipe

Composition: Any sort of data transformation that we do should be obvious.

Compossibility: System design principle that deals with the relationship between components. We can move things or functionality around just like in a factory.

* Compose is one of the most powerful concepts in JS.
* Compose is not native to JS. But very common so can use a library. Ramdajs

Pipe: Essentially same as compose just reverse.

compose(fn1, fn2, fn3)(-50) right to left

pipe(fn3, fn2, fn1)(-50) left to right.

17. Arity

Arity: The number of arguments that a function takes.

* Not a solid rule but usually a good practice to have less amount of arguments. The lesser the better to use.
* We can use curry, compose, pipe
* It help us to make our code more reusable.

18. Is FP The Answer to Everything

* If multiple things in our program that handle some piece of data then it becomes very buggy and complicated.
* Good for distributed system, parallelism.

20. Reviewing FP



* A function without a return statement is just a procedure.
* FP is good at one to one data transformation.
* JS allows us:
  + to assign fn to variables.
  + pass fn to another fn.
  + return fn from fn.
  + compose fn.
  + pipe fn.
* Immutable concepts like primitive types, using map, concate on arrays, Object.assign, cloning obj with spread operator.
* Trying to have pure fn, trying to avoid mutating state to have predictable code.