Tech Presentation

Functional Programming

Table of Contents

Points for discussion

WHY

"Any fool can write code that a computer can understand. **Good programmers** write code that humans can understand."

— Martin Fowler

WHY

"We live in an age of the crisis of software complexity and distributed large systems have become so complex that simple procedural OOP can't be used to reason about it and hence FP is a very promising solution for this"

— Eric Meyer

WHY

Importance of Functional Programming

WRITING SIMPLE CODE

DETERMINISTIC CODE

HIGHLY SCALABLE & RE USABLE CODE

NO SIDE EFFECTS

EASILY TESTABLE CODE

WHAT

What is Functional Programming

a paradigm

(mindset/worldview)

WHAT

What is Functional Programming

"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 the application state flows through pure functions. "

— Eric Elliot

HOW

Functional Programming Concepts

- Avoid Side Effects

HOW

Functional Programming Concepts

- Avoid Side Effects
- Avoid Mutations

HOW

Functional Programming Concepts

- Avoid Side Effects
- Avoid Mutations
- Avoid Shared State

HOW

Functional Programming Concepts

- Avoid Side Effects
- Avoid Mutations
- Avoid Shared State
- Use Pure Functions

HOW

Functional Programming Concepts

- Avoid Side Effects
- Avoid Mutations
- Avoid Shared State
- Use Pure Functions
- Use Function Composition

HOW

Functional Programming Concepts

- Avoid Side Effects
- Avoid Mutations
- Avoid Shared State
- Use Pure Functions
- Use Function Composition
- Use Declarative Code instead of Imperative Code

Pure Functions

A pure function is a function which:

- Given the same input, will always return the same output.
- Produces no side effects.

Pure Function Example

```
function sum(a,b){
return a+b
}
```

Pure Functions Rules

1. Have Input Parameters
2. No Stateful Values
3. return based on Input
4. No Side Effects

Side Effects

A Side Effect is an observable change outside the function

```
let count = 0
function increment(){
count++
return count
}
```

Side Effects makes code

- Difficult to Predict
- Harder to Debug
- Difficult to reason about

Side Effects

- Changing a value globally (variable or property of any data structure)
- Changing the original value of a function object
- Printing to screen or Logging.
- Trigerring external process
- Invoking other functions that have side effects
- Saving or retrieving something database
- writing and reading files
- Interacting and changing the DOM

```
var x = 1
fun1()
console.log(x)
fun2()
console.log(x)
fun3()
console.log(x)
fun4()
console.log(x)
```

State

A program is considered stateful if it is designed to remember data from events or user interactions. The remembered information is called the state of program.

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 scope.

— Eric Elliot

Shared State Example

```
var currentUser = 0
var users = [
{name : 'faiz', weight : 85},
{name : 'khan', weight : 68}
]

const AddWeight = (weight){
users[currentUser].weight += weight
}
```

Avoiding Mutations

Mutable -> changable

Immutable -> Not Changable

objects in Java Script are mutable

JavaScript Concepts

- Higher Order Functions
- Arrow functions
- Closure

Higher Order Functions

- map
- filter
- reduce

Arrow Functions

```
function sum(a,b) {
return a+b
}
```

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

Closure

Closure is the property through which the inner functions have access to the outer function variables and arguments even when the outer function has returned.

Closure Example

```
function sum(a) {
return function(b) {
  return a+b
                    }
}
```

Currying

Currying is the technique through which functions with multiple arguments are transformed into a series of nested functions with less number of arguments.

Currying Example

Without Currying

```
function mult(a, b, c, d) {
return a * b * c * d;
}
mult(1, 2, 3, 4); // 24
```

With Currying

```
function mult(a) {
return function (b) {
return function (c) {
return function (d) {
return a * b * c * d;
};};};}
mult(1)(2)(3)(4); // 24;
```

Function Composition

Combining of functions for the purpose of producing a new function or performing some computation

Procedure

A procedure is a collection of functionality, it may have inputs or may not it may have return values or may not and it frequently does its work by working on a shared state.

Function

- Functions have an input
- Functions return a value
- Functions are simplified to a single task