Agenda

1. Pure Functions
2. First-Class Objects
3. Higher-Order Functions
4. Composition
5. Immutability
6. Declarative vs. Imperative
7. Benefits of Functional Programming

What Is Functional Programming?

Functional programming is a programming paradigm designed to handle pure mathematical functions. This paradigm is totally focused on writing mo compounded and pure functions.

Functional programming is the process of building software by composing **pure functions** , avoiding **shared state, mutable data,** and **side-effects**.

Pure Functions

As a software programmer/developer, you write source code to produce an output based on the inputs. Usually, you write functions to perform the tasl based on inputs and produce an output. We need to make sure these functions are,

- **Predictable**: It produces a predictable output for the same inputs.
- **Readable**: Anyone reading the function as a standalone unit can understand its purpose completely.
- **Reusable**: Can reuse the function at multiple places of the source code without altering its and the caller's behaviour.
- **Testable**: We can test it as an independent unit.

A Pure Function has all the above characteristics. It is a function that produces the same output for the same input. It means it returns the same resu when you pass the same arguments. A pure function shouldn't have any side effects to change the expected output.

Example 1 :

```
function sayGreeting(name){
return `Hello ${name}`;
}
```

Example 2:

```
let greeting = "Hello";
function notPureGreeting(name){
return `${greeting} ${name}`;
}
```

The function ***sayGreeting()*** is a pure function.

Can you guess why?

It is a pure function because you always get a ***Hello*** as output for the pass as an input.

Function ***notPureGreeting*** is not a pure function because the function's output now depends on an outer state called greeting.

What if someone changes the value of the greeting variable to ***Hola***?

It will change the output of the ***notPureGreeting()*** function even when you pass the same input.

One of the reasons that pure functions are so powerful is that they are way easier to test and debug. Since a pure function should return the same outp given specific input value, we can test this function very easily.

First-Class Objects

Functions in JavaScript are first-class objects, also known as "*first-class citizens*". This means that we can work with functions in JavaScript the sam way as variables.

- In fact, JavaScript functions are themselves types of objects.

- A first-class function can thus be expected to support the same operations we would expect from other objects.
- So what are these operations?

Generally, first-class objects can:

1. Be stored in a variable;
2. Be passed as arguments to functions;
3. Be returned by functions;
4. Be stored in some data structure; and,
5. Hold their own properties and methods.

Storage in a Variable

```
// Function definition and invocation
function speak(string) {
  console.log(string);
}
speak("Hello");                    // logs "Hello"

// Store in a variable
var talk = speak;
talk("Hi");                        // logs "Hi"
```

We declare our function `speak()` and subsequently demonstrate its invocation. So far things are looking pretty standard. Then, we define variable `talk` and assign in the value of `speak`. When we call `talk` as a function we get the same functionality as when calling `speak`! Th shows that you can store a function in a variable. Not only that, but in this case both `speak` and `talk` refer to the same function, and th expression `speak === talk` will return `true`

Passage as an Argument to a Function; Return Value from a Function

```
// Pass as an argument to a function
// Return from a function
function functionReturner(fn) {
  return fn;
}
var chat = functionReturner(talk);
chat("Good Morning");              // logs "Good Morning"
```

We declare another function, self-consciously called `functionReturner`, which serves the somewhat silly purpose of receiving a function as a argument and simply returning it. And indeed, when we pass `functionReturner` our function `talk` as an argument and assign its return value to th variable `chat`, we get exactly the same functionality as we did with `talk` or `speak`. This is a very arbitrary example, but it shows that functions ca indeed be passed as arguments to other functions, and that they can be the return value from other functions too— just like any other first-class object.

Storage in a Data Structure

```
// Function definition and invocation
function speak(string) {
  console.log(string);
}
speak("Hello");                    // logs "Hello"

// Store in a variable
var talk = speak;
talk("Hi");                        // logs "Hi"

// Store in a data structure
var myFuncs = [talk];
myFuncs[0]("Good Afternoon");
```

Functions can also be stored in data structures. In this example, we store our function `talk` inside an array called `myFuncs`. When we attempt access `myFuncs[0]`, which we expect to be a function, and call it, we do in fact get the functionality we are expecting. A function can similarly be store on other data structures, such as an object, in which case we generally refer to it as a *method* of that object.

Owner of a Property

```
// Function definition and invocation
function speak(string) {
  console.log(string);
}
speak("Hello");                    // logs "Hello"

// Store in a variable
var talk = speak;
talk("Hi");                        // logs "Hi"
// Owns properties
talk.myProperty = "bananas";
console.log(talk.myProperty);      // logs "bananas"
```

We assign our function `talk` an arbitrary property called `myProperty` and store the value `"bananas"` inside it. Just like with any other object, w can directly access this property.

Uses of a First-Class Function

So, we know that functions in JavaScript are first-class citizens. Now what do we do with that knowledge? . First-class functions give us a wide variety flexible and powerful design patterns. These patterns let us write more readable, more dynamic, and more concise code. Let's take a look at a few.

Higher Order Function

A ***higher-order function*** is one that either has a function as a parameter, or returns a function.

In other words, higher-order functions do work on other functions. The classic example of higher-order functions are built-in functions that we use eve day to manipulate JavaScript objects and data structures. These built-in functions do some kind of work such as iteration or transformation. Often, w supply such built-in functions with anonymous functions as callback arguments, but we can also pass in existing functions.

Consider the following:

```
var myNums = [1, 2, 3, 4, 5];

function doubleNum(num) {
  return num * 2;
}

// Built-in Array.prototype.map function, using anonymous function argument
var doubledNums = myNums.map(function(num) {
  return num * 2;
});
console.log(doubledNums);          // logs "[2, 4, 6, 8, 10]"

// Built-in Array.prototype.map function, using named callback argument
var otherDoubledNums = myNums.map(doubleNum);
console.log(otherDoubledNums);     // logs "[2, 4, 6, 8, 10]"
```

In this snippet, we first define a variable `myNums` which contains an array of numbers. We also define a function `doubleNum` which accepts a numb as an argument and returns double the value of that number. We then demonstrate higher-order functions using built- function `Array.prototype.map` , which iterates over an array, transforming each value and returning a new array with those values. In the fir example, we provide `map` with an anonymous function that doubles each value in the array. We pass `map` our predefined function `doubleNum` which it then uses in the transformation process. In both cases we get the same result.

Asynchronous Function

Another common use of first-class function is when an asynchronous callback is necessary. This is often the case in web applications where som function must wait to be called until a server has returned some value or a promise has been fulfilled. We won't get into that here but we can simula delaying a function call using the built-in function setTimeout, which accepts a callback and a delay as values. Here is an example using our trusty spea function.

```
function speak(string) {
  console.log(string);
}
```

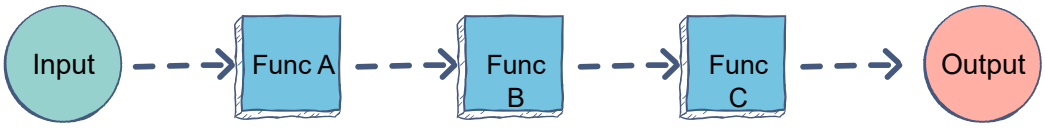
```
var delayedFunction = function(fn) {
  return function(val, delay) {
    setTimeout(function() {
      fn(val);
    }, delay);
  };
};

var delayedSpeak = delayedFunction(speak);
delayedSpeak("I'm late!", 1000);      // logs "I'm late" after a 1 second delay
```

We define a new higher-order function called `delayedFunction`, which accepts a function as a an argument and then returns a new function, whic accepts a value and a delay. The new function uses `setTimeout` to wait `delay` length before executing the original function we provide to `delayedFunction` with the provided value as its argument. We then use `delayedFunction` passing in our `speak` function, to create a ne function called `delayedSpeak`. Now we can use `speak` with some delay! Here we have not only used first-class functions, and higher-order function but also seen how to asynchronously execute our code.

Function composition

Function composition is an approach where the result of one function is passed on to the next function, which is passed to another until the fin function is executed for the final result. Function compositions can be composed of any number of functions.



Traditional approach

Traditionally, function composition follows this format:

```
const double = x => x * 2
const square = x => x * x
// Tradition approach
var output1 = double(2);
var output2 = square(output1);
console.log(output2);
// variant two
var output_final = square(double(2));
console.log(output_final);
```

In the code above, we can see that we need to call the *`double`* *function* followed by the *`square`* *function* to square a term that has been doubled. W can do this by either assigning the values individually in variables and calling functions onto them , or we could use a more direct approach.

Alternate approach

Another approach is to use the `compose` and `pipe` functions.

`compose` function

`compose` function takes any number of functions and invokes them all one after the other:

```
const compose = (...fns) => x => fns.reduceRight((y, f) => f(y), x);
const double = x => x * 2
const square = x => x * x

// function composition
var output_final = compose(square, double)(2);
console.log(output_final);
```

In the code above, we can see that the `compose` function is implemented using a general approach , so now it can take any number of functions. Th output remains the same as before even with using this implementation.

`pipe` function

On the other hand, we can reverse the order of the function invocation by using the `pipe` function:

```
// function composition using pipe of any number of functions
const pipe = (...fns) => x => fns.reduce((y, f) => f(y), x);
const double = x => x * 2
const square = x => x * x

// function pipe
var output_final = pipe(square, double)(2);
console.log(output_final);
```

In the code above, we can see that the `pipe` function is implemented using a general approach, so it can now take any number of functions. This similar to the previous `compose` function, except that it uses `reduce` instead of the `reduceRight` method. The output is different in this cas because the `square` function is invoked before the `double` function while, in our `compose` function, it was in the opposite order.

Immutability in JavaScript

In one translation, it can mean *unchanging* — however, setting something up as a ***let*** or ***var*** and then not allowing it to change might as well be as goo as using ***const.***

There are a lot of people who take this translation a bit too far in its application, and as a result, misunderstand the concept and depth of immutability.

In this piece, we'll be decrypting the idea of immutability and how it is applied in JavaScript.

What exactly is immutability?

When it comes to object-oriented and functional programming, the concept of immutability is applied at the object level.

It's to do with the state and how you're not allowed to modify after it's been created. Every language has its own system of state management and befo we go any further, we need to understand exactly what a state is.

In object-oriented, a state has two parts to it — the properties and the values. The properties are generally static, meaning that they don't change. Th values, however, are expected to be dynamic and therefore changeable.

So if we were to take a step back and observe the state, it is essentially the 'shape' and 'appearance' of a particular object at any given point in time.

What this means that when properties are added or removed, the overall shape, and therefore state, changes.

Immutability in JavaScript

- In JavaScript, all primitive types are weakly immutable by design.
- Because the shape of undefined , null, boolean, number, bigInt , String and Symbol is singular and flat. It is a single assignment and when a primitive type is initialised, there is a clear understanding of what it's going to look like.
- It's not going to shrink, expand, or do funny things if you change the assigned value. It might change your processed output when popped into a function, but that's a different story.
- If you need it to be strongly immutable, use const.
- In JavaScript, custom objects are highly mutable — unless you explicitly tell it not to be.
- How do you do this? You can either do it via Object.defineProperty() or Object.freeze()

Example :

```
const updateLocation = (data, newLocation) => {
    return {
      Object.assign({}, data, {
        location: newLocation
    })
  }
}
```

- By using *Object.assign()*, we can create a function that does not mutate the object passed to it.
- This will generate a *new* object instead by copying the second and third parameters into the empty object passed as the first parameter.
- Then the new object is returned.

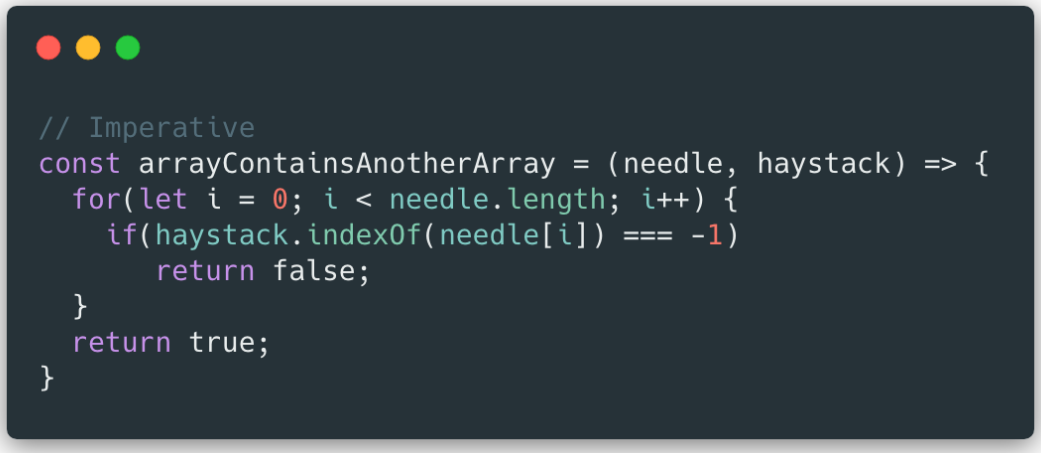
Note: *updateLocation()* is a pure function. If we pass in the first user object, it returns a new user object with a new value for the location.

Declarative vs. Imperative

Functional programming is **declarative** rather than **imperative.**

- Functional Programming is a declarative programming paradigm, in contrast to imperative programming paradigms.
- Declarative programming is a paradigm describing **WHAT** the program does, without explicitly specifying its control flow.
- Imperative programming is a paradigm describing **HOW** the program should do something by explicitly specifying each instruction (or statement) step by step, which mutate the program's state.
- This "**what vs how**" is often used to compare both of these approaches because... Well, it is actually a good way to describe them.
- Granted, at the end of the day, everything compiles to instructions for the CPU. So in a way, declarative programming is a layer of abstraction on top of imperative programming.

The Imperative Code



```
// Imperative
const arrayContainsAnotherArray = (needle, haystack) => {
  for(let i = 0; i < needle.length; i++) {
    if(haystack.indexOf(needle[i]) === -1)
       return false;
  }
  return true;
}
```

Let's break down the thought process required to figure out what's going on here.

1. JavaScript isn't typed, so figuring out the return and argument types is the first challenge.

2. We can surmise from the name of the function and the two return statements that return literal boolean values that the return type is boolean.

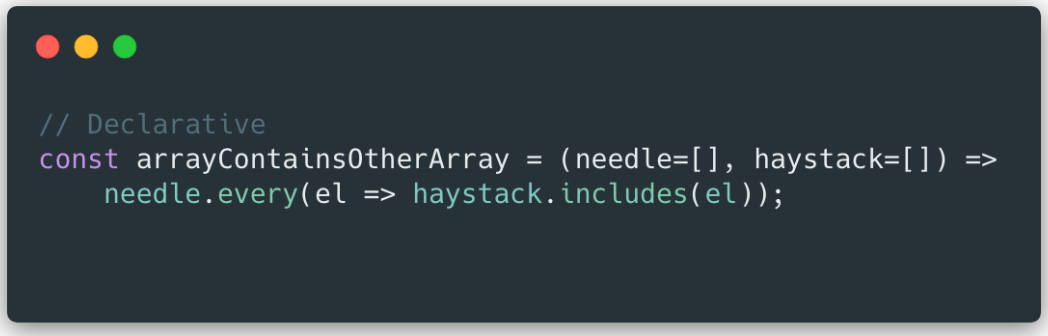
3. The function name suggests that the two arguments may be arrays, and the use of **needle.length** and **haystack.indexOf** confirms that.

4. The loop iterates the **needle** array and exits the function returning false whenever the currently indexed value of the **needle** array is not found in the **haystack** array.

5. If the loop completes without exiting the function, then we found no mismatches and true is returned.

6. Thus, if all the values of the **needle** array (in any order) are found in the **haystack** array, we get a true return, otherwise false.

The Declarative Code



```
// Declarative
const arrayContainsOtherArray = (needle=[], haystack=[]) =>
    needle.every(el => haystack.includes(el));
```

That took fewer lines, but you still have to break it down to understand what it's doing. Let's see how that process differs.

1. JavaScript isn't typed, so figuring out the return and argument types is the first challenge.
2. We can surmise from the name of the function and the returned result of an array's **every** method that the return type is boolean.
3. The function name suggests that the two arguments may be arrays, as do the default values now added to the arguments for safety.
4. The **needle.every** call names its current value '**el**', and checks if it is present in the **haystack** array using **haystack.includes**.
5. The **needle.every** call returns true or false, telling us, quite literally, whether every element in the **needle** array is included in the **haystack** array.

Note: Here is the link where you can learn more about Imperative and Declarative programming: https://medium.com/weekly-webtips/imperative-v declarative-programming-in-javascript-25511b90cdb7

Benefits of Functional Programming

You are writing code to build an application. In development journey, you want to reuse the code of a few lines (100) at different places and f applications functions are helpful. We can write functions at one place and we will be able to access those functions from anywhere in the prograr Functional programming has the following features —

1. Reduces code redundancy.
2. Improves modularity.
3. Helps us to solve complex problems.
4. Increases maintainability.

Note: Here is the link where you can learn more about the benefits of Functional Programming: https://www.unthinkable.co/blog-post/7-unbeatabl advantages-of-functional-programming/

- Today's Task :

Write a Blog of about 300 words explaining about the difference between imperative and declarative programming and benefits of function programming based on the knowledge you got by reading the blog's those links has been provided in the topics.

Interview Questions

> Explain Higher Order Functions in JavaScript.

Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions. Highe order functions are a result of functions being **first-class citizens** in JavaScript.

Examples of higher-order functions:

```
function higherOrder(fn) {
  fn();
}

higherOrder(function() { console.log("Hello world") });
```

```
function higherOrder2() {
  return function() {
    return "Do something";
  }
}
var x = higherOrder2();
x()   // Returns "Do something"
```

> What is functional programming?

Functional programming is a paradigm in which programs are built in a declarative manner using pure functions that avoid shared state and mutab data. Functions that always return the same value for the same input and don't produce side effects are the pillar of functional programming. Mar programmers consider this to be the best approach to software development as it reduces bugs and cognitive load.

- Cleaner, more concise development experience

- Simple function composition
- Features of JavaScript that enable functional programming (`.map` , `.reduce` etc.)
- JavaScript is multi-paradigm programming language (Object-Oriented Programming and Functional Programming live in harmony)

> What is a pure function?

A **pure function** is a function which:

- Given the same inputs, always returns the same output, and
- Has no side-effects