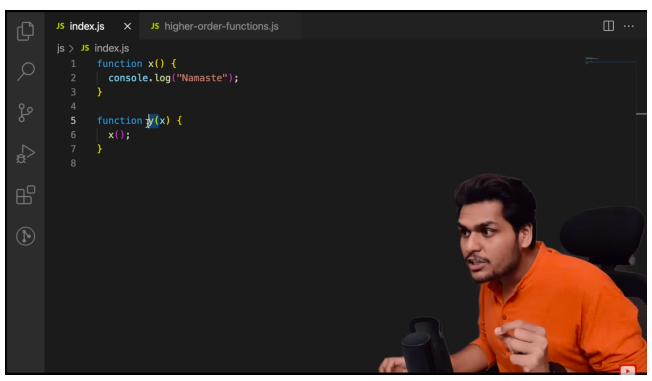
Higher Order Functions ft. Functional Programming

Sunday, 20 August 2023 12:27 AM

→ Higher order functions: Functions that take in another fⁿ as argument or returns a fⁿ from it are higher order fⁿs.

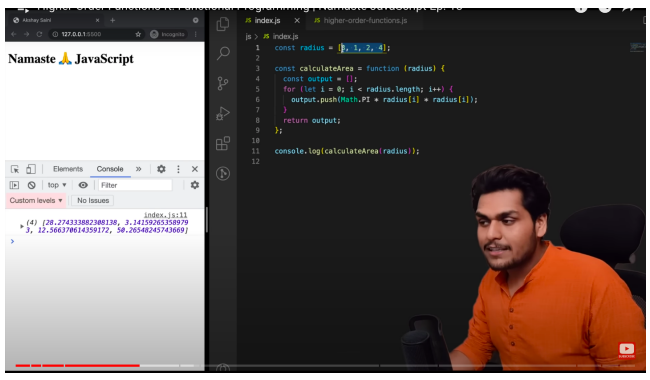


→ Here y is a higher order fⁿ.
→ And x is a callback fⁿ.

→ Y() is a higher order fⁿ as it is taking another fⁿ as an argument.

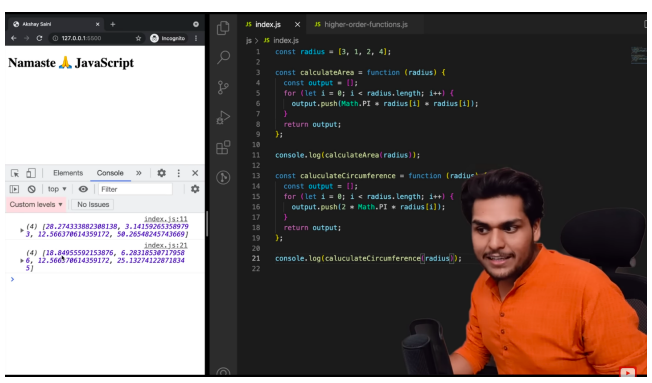
→ X() is a callback fⁿ because it is being passed in another fⁿ as an argument.

→ Mistakes to avoid in an interview:



→ Program to calculate area of a circle

Now, if someone tells us to write a program to calculate the circumference of a circle, we might do this:

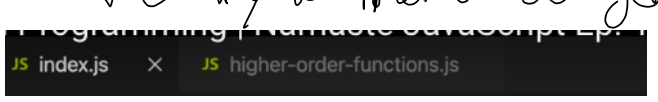


→ Program to calculate circumference of a circle.

So, what we are doing is that we are repeating ourselves again & again which is not recommended.
There is a principle in software engineering: DRY principle (Do not repeat yourself).

→ How to optimize the above code?

We try to make a generic fⁿ which can do everything.



```
js > JS index.js
1   const radius = [3, 1, 2, 4];
2
3   const area = function (radius) {
4     return Math.PI * radius * radius;
5   };
6
7   const cicumference = function (radius) {
8     return 2 * Math.PI * radius;
9   };
10
11  const calculate = function (radius, logic) {
12    const output = [];
13    for (let i = 0; i < radius.length; i++) {
14      output.push(logic(radius[i]));
15    }
16    return output;
17  };
18
19  console.log(calculate(radius, area));
20  console.log(calculate(radius, cicumference));
21
```

→ This code is better because we have abstracted the logic into smaller fn's & each unit has its own responsibility.
→ Also we are not repeating ourselves.
→ So this is Functional programming where we think all logic into small reusable components which are functions.

→ This increases reusability and code modularization (which is dividing the code into small independent modules)
→ Here we can see that calculate is a higher order fn & area, circumference are callback fn's.
→ In above code, we pass radius & a callback fn to calculate(). And in line 14 we send value of radii to the callback fn's (which can be any callback fn).

Note:

```
const calculate = function (arr, logic) {
  const output = [];
  for (let i = 0; i < arr.length; i++) {
    output.push(logic(arr[i]));
  }
  return output;
};

console.log(radius.map(area));

console.log(calculate(radius, area));
```

We can see that we've made a fn which behaves just as .map() does which is provided by JS.

But we can't use it like .map(), so in order to do that, we can declare the fn with Array.prototype and use it just like map().

```
Array.prototype.calculate = function (logic) {
  const output = [];
  for (let i = 0; i < this.length; i++) {
    output.push(logic(this[i]));
  }
  return output;
};

console.log(radius.map(area));

console.log(radius.calculate(area));
```

This is therefore, the polyfill for map() fn.

In this, 'this' points to the array.
This polyfill takes in a callback fn for logic & also returns a new array just like map().