Currying

Thursday, 24 August 2023 1:24 PM

→ Currying is the process of taking a f^n with multiple arguments & turning it into a sequence of f^ns each with only a single argument.

We can curry a f^n using 2 methods:
1) bind() method 2) closures

1) Using bind() method:

```
let multiply = function (x, y) {
  console.log(x * y);
}

let multiplyByTwo = multiply.bind(this, 2);
multiplyByTwo(5);
```

→ This essentially means that we've set the value of 'x' to be 2 forever when using multiplyByTwo() method.

→ So, we've curried the f^n multiply() into a f^n that just takes 1 argument.

→ So, whatever argument we pass to multiplyByTwo() becomes the value for 'y'.

→ So, multiplyByTwo() method has basically become this:

```
let multiplyByTwo = function (y) {
  let x = 2;
  console.log(x * y);
}
```

2) Using closures:

```
let multiply = function (x) {
  return function (y) {
    console.log(x * y);
  }
}

let multiplyByTwo = multiply(2);
multiplyByTwo(3);
```

→ So here multiply(2) will pass argument as '2' & 'x' becomes 2.

→ Now multiply() f^n returns an anonymous f^n which is basically a closure i.e it is a f^n bundled with its lexical environment; so it will have access to 'x' which in its parent's memory.

→ Thus, we have curried multiply() into a f^n that just takes 1 argument i.e multiplyByTwo().

Note: Basically with currying, we turn f(a,b) to f(a)(b), so we turn a single f^n taking multiple arguments to a sequence of f^ns taking a single argument.

```
function f(a) {
  return function (b) {
    return `${a} ${b}`;
  };
}
```

(basic eg of currying)

```
console.log(f(5)(6));
```

Q: Why should we use currying?

ans:

- ✅ It makes a function pure which makes it expose to less errors and side effects.
- ✅ It helps in avoiding the same variable again and again.
- ✅ It is a checking method that checks if you have all the things before you proceed.
- ✅ It divides one function into multiple functions so that one handles one set of responsibility.

→ A pure f^n is a f^n which returns the same value when same arguments are passed.

→ Converting a simple f^n to a curried f^n:

```
/*Simple function*/
const add = (a, b, c)=>{
    return a+ b + c
}
console.log(add(1,2 ,3)); // 6

/* Curried Function */
const addCurry = (a) => { // takes one argument
    return (b)=>{                //takes second argument
        return (c)=>{            //takes third argument
            return a+b+c
        }
    }
}
console.log(addCurry(1)(2)(3)); //6
```

→ We can make a general f^n & then store that f^n for a particular value inside a variable, so that, the new f^n always performs the same thing & can be reused wherever we want it.

```
function evaluate(operation) {
  return function (a) {
    return function (b) {
      if (operation === "sum") return a + b;
      else if (operation === "multiply") return a * b;
      else if (operation === "divide") return a / b;
      else if (operation === "substract") return a - b;
      else return "Invalid Operation";
    };
  };
}

const mul = evaluate("multiply");

console.log(mul(3)(5)); // 15
console.log(mul(2)(6)); // 12
```

→ So we store evaluate('multiply') inside mul.
So now, mul will always perform multiplication.

→ Infinite Currying:

Implement an add() f^n which is flexible & can take in 'n' no. of arguments and return their sum.

```
function add(a) {
  return function (b) {
    if (b) return add(a + b);
    return a;
  };
}

console.log(add(5)(2)(4)(8)());
```

→ So this is basically infinite currying.
→ In the returned f^n if we have the value of 'b' then we run the f^n again, otherwise we return the value of 'a' which will be the sum obtained in the previous iteration.

→ add(5) returns us a f^n which takes in an argument. If we console add(5), we'll get the returned f^n as o/p.

→ That returned f^n is called with the value of 'b' (here, 2), since 'if' condition is satisfied, thus we pass (a+b) as argument to add().

So add(7) will again return a f^n that will be called with value of b = 4,
Again the process repeats & (a+b) is sent as an argument to add().
So add(11) will again return a f^n that will be called with value of 'b = 8'.

→ Finally add() receives '19' as argument & it returns a f^n, now that f^n doesn't receive value of 'b', so it returns 'a' which is '19' as value of 'a' previously received was '19'.
→ Thus, we've achieved infinite currying.

→ Currying vs Partial Application:

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

const x = sum(10);
console.log(x(5, 6));
console.log(x(3, 2));

// or

console.log(sum(20)(1, 4));
```

→ eg of partial application.
→ Currying changes a f^n that takes multiple arguments to a sequence of f^ns that take single arguments.
→ Partial application changes a f^n that takes multiple arguments to a f^n that has small arity i.e f^n with less arguments.

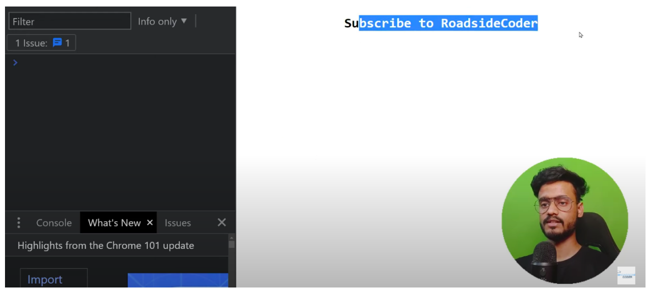
→ Real world implementation of currying using DOM manipulation:

```
function updateElementText(id) {
  return function (content) {
    document.querySelector("#" + id).textContent = content;
  };
}

const updateHeader = updateElementText("heading");

updateHeader("Subscribe to RoadsideCoder");
```

(f^n to update heading content)



(o/p)

→ Assuming there is an <h1> tag with id="heading".
→ Now, we can have a f^n updateElementText() that takes in an id & returns a f^n that selects that element & changes its content.
→ So we curry the f^n by providing id to it & store it as a closure.
→ Then we can reuse the f^n however & wherever we want & update its content.

Q: Write a curry implementation that converts f(a, b, c) to f(a)(b)(c).
Ans:

→ Polyfill of curry():

```
function curry(func) {
```

→ So our f^n curry() takes in a callback f^n

```javascript
  return function curriedFunc(...args) {
    if (args.length >= func.length) {
      return func(...args);
    } else {
      return function (...next) {
        return curriedFunc(...args, ...next);
      };
    }
  };
}

const sum = (a, b, c) => a + b + c;

const totalSum = curry(sum);

console.log(totalSum(1)(2)(3));
```

→ So, our curry() takes in a callback and converts it to a curried f^n & returns it.

→ The returned f^n takes the arguments & checks if length of arguments is greater than or equal to the func.length, which is basically the no. of parameters the callback f^n can accept (in this case, 3).

→ For the first time our args.length will be '1', so else condition will run (since '1' < '3'), it will return a f^n which recursively calls curriedFunc() with current argument & the next argument.

→ We again run the curriedFunc(), now if condition is false again (as, '2'<'3') so again we will return a f^n which will take '3' as argument and when called it will recursively call the curriedFunc() with previous & current arguments.

→ Now, as curriedFunc() is called, if condition will execute (as "3' = '3') and it will finally return the callback f^n which is transformed to a curried f^n and it will spread the array & take the arguments individually.