Pure Functions

Pure vs. Impure Functions

– Pure functions take some input and give a fixed output. Also, they cause no side effects in the outside world.

```
const add = (a, b) => a + b;
```

– Here, `add` is a pure function. This is because, for a fixed value of `a` and `b`, the output will always be the same.

```
const SECRET = 42;
const getId = (a) => SECRET * a;
```

– `getId` is not a pure function. The reason being that it uses the global variable `SECRET` for computing the output. If `SECRET` were to change, the `getId` function will return a different value for the same input. Thus, it is not a pure function.

```
let id_count = 0;
const getId = () => ++id_count;
```

– This is also an impure function, and that too for a couple of reasons:
  i. it uses a non-local variable for computing its output, and
  ii. it creates a side effect in the outside world by modifying a variable in that world.

– This can be troublesome if we had to debug this code.
– What's the current value of `id_count`? Which other functions are modifying `id_count`? Are there other functions relying on `id_count`?
– Because of these reasons, we only use pure functions in functional programming.
– Another benefit of pure functions is that they can be parallelized and memoized. Have a look at the previous two functions. It's impossible to parallelize or memoize them. This helps in creating performant code.

The Tenets of Functional Programming

– So far, we have learned that functional programming is dependent on a few rules. They are as follows.
  - Don't mutate data
  - Use pure functions: fixed output for fixed inputs, and no side effects
  - Use expressions and declarations
– When we satisfy these conditions, we can say our code is functional.

Functional Programming in JavaScript

– JavaScript already has some functions that enable functional programming. Example: `String.prototype.slice`, `Array.protoype.filter`, `Array.prototype.join`.
– On the other hand, `Array.prototype.forEach`, `Array.prototype.push` are impure functions.
– One can argue that `Array.prototype.forEach` is not an impure function by design but think about it – it is not possible to do anything with it except **mutating** non-local data or **doing** side effects. Thus, it is okay to put it in the category of impure functions.
– Also, JavaScript has a `const` declaration, which is perfect for functional programming since we will not be mutating any data.

Pure Functions in JavaScript

– Let us look at some of the **pure functions** (methods) given by JavaScript.

Filter

As the name suggests, this filters the array.

```
array.filter(condition);
```

The condition here is a function that gets each item of the array, and it should decide whether to keep the item or not and return the truthy boolean value for that.

```
const filterEven = x => x%2 === 0;
[1, 2, 3].filter(filterEven);
// [2]
```

Notice that `filterEven` is a pure function. If it had been impure, then it would have made the entire filter call impure.

Map

`map` maps each item of array to a function and creates a new array based on the return values of the function calls.

```
array.map(mapper)
```

`mapper` is a function that takes an item of an array as input and returns the output.

```
const double = x => 2 * x;
[1, 2, 3].map(double);
// [2, 4, 6]
```

Reduce

`reduce` reduces the array to a single value.

```
array.reduce(reducer);
```

`reducer` is a function that takes the accumulated value and the next item in the array and returns the new value. It is called like this for all values in the array, one after another.

```
const sum = (accumulatedSum, arrayItem) => accumulatedSum + arrayItem
[1, 2, 3].reduce(sum);
// 6
```

Concat

`concat` adds new items to an existing array to create a new array. It's different from `push()` in the sense that `push()` mutates data, which makes it impure.

```
[1, 2].concat([3, 4])
// [1, 2, 3, 4]
```

You can also do the same using the spread operator.

```
[1, 2, ...[3, 4]]
```

Object.assign

`Object.assign` copies values from the provided object to a new object. Since functional programming is predicated on immutable data, we use it to make new objects based on existing objects.

```
const obj = {a : 2};
const newObj = Object.assign({}, obj);
newObj.a = 3;
obj.a;
// 2
```

With the advent of **ES6**, this can also be done using the spread operator.

```
const newObj = {...obj};
```

Creating Your Own Pure Function

– We can create our pure function as well. Let us do one for duplicating a string `n` number of times.

```
const duplicate = (str, n) =>
      n < 1 ? '' : str + duplicate(str, n-1);
```

– This function duplicates a string `n` times and returns a new string.

```
duplicate('hooray!', 3)
// hooray!hooray!hooray!
```

Assignment

Write a pure recursive function for the following algorithm:

To calculate the factorial of n:

```
1.    If n = 0:
            1.1.  Terminate with answer 1.
2.    If n * 0:
            2.1.  Let f be the factorial of n–1.
            2.2.  Terminate with answer (n x f).
```