Functional Programming concepts in TypeScript

Functions are first class citizens

JavaScript is a functional programming language in that sense that functions are truly first class citizens. A first-class citizen is an entity which supports all the operations generally available to other entities. These operations typically include being passed as an argument, returned from a function, and assigned to a variable.

For that reason, we can write that function:

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

as a funcional expression:

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

It creates an object of type (any, any) -> any.

Currying and Partial Function Application

Apply a function partly is a very powerful technique.

A simple example is the previous function `add`. It would be nice if we could use `add` like:

```
var res2 = add(1)(3); // => 4

var add10To = add(10);
var res = add10To(5); // => 15
```

If we can use the add function with that two signatures:

```
add:: number -> number -> number

or

add:: number -> (number -> number)
```

we must to change its implementation:

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

a function that returns another function. Now we can use both signatures. The second one allow us to compose functions, applying a function partially.

Same function can be defined as follow:

```
const add = (x: number) => (y: number) => x + y;

console.log(add(123)(123)); // --> returns 246
```

Obviously, we can apply generics to do more reusable:

```
function add<T>(x: T) {
  return (y: T) => x + y;
}

console.log(add(123)(123)); //-> output: 246
console.log(add("123")(123)); //-> output: "123123"
```

We can use one more another notation, based on Arrow functions:

```
const updateStateAndVisibility = obj =>
         state =>
          visibility =>
           Object.assign({}, obj, state, visibility);

const value = updateStateAndVisibility({type: 'camera'})({state:
'online'})({visible: true});
console.log(value);
```

The result is:

```
{
  state: "online",
  type: "camera",
  visible: true
}
```

Function Composition

We can compose multiple functions to get a final result. That powerful technique allows us to generalize functions. We can do that:

```
//Compose function
function compose(...args, funct) {
 return args.reduce((a,b) => funct(a)(b));
}

//add function
function add(a) {
 return (b) => a + b;
}

//we can execute that:
console.log(compose([1,2,3], add));  //-> result is 6
console.log(compose(["a", "b", "c"], add)); //-> result is "abc"
```

High Order Functions

A High Order Function is a function that does at least one of the following:

- takes one or more functions as arguments,
- returns a function as its result.

Are also known as operators.

Most of the Array's functions are High Order Functions, for example, map or filter.

We can create a High Order Function:

repeat function as example, with this signature:

```
(times, funct, initial) => any;
```

parameters:

- times: a number as a repeat limit.
- funct: function that will be applied.
- initial: initial value for the funct function.

the function declaration:

```
const repeat = (times, funct, initial) => times > 0 ? repeat(times - 1, funct,
funct(initial)) : funct(initial);
```

With that function we can do something like that:

```
console.log(repeat(5, (_) => _ + '*', ''));
```

and we get as output: "*****"

Combining functions and using High Order functions we can do complex functionality with small pieces of code. Write a small pieces of code allows as to reuse more and do less bugs.

NOTE: all code can be found here