1. Functional Programming

1.2 Functional Concepts

1.2.1 Immutability

To mutate is to change, so to be immutable is to be unchangeable. In functional programming, data is immutable. It never changes. Then, how data is modified? Instead of changing the original data structures, we build changed copies of those data structures and use them instead.

Consider and object that represents the color lawn:

```
let color_lawn = {
    title: "lawn",
    color: "#00FF00",
    rating: 0
}
```

We can build a function that would rate colors, and use that function to change that rating of color object:

In the following example, the data which reference is given as parameter is effectively changed, which is bad, because the function changes or mutate the original data.

```
function rateColor(color, rating) {
    color.rating = rating
    return color
}
consle.log(rateColor(color_lawn, 5).rating)
console.log(color_lawn.rating)
```

In the following example, we rewrite the function so that is doesn't harm the original object:

```
var rateColor = function(color, rating) {
    return Oject.assign({}, color, {rating: rating})
}
console.log(rateColor(color_lawn, 5).rating) /// prints 5
console.log(color_lawn.rating) /// prints 0
```

We can write the same function using the ES6 arrow function along with the ES7 spread operator. The function uses the spread operator to copy the color into a new object and then overwrite its rating:

```
const rateColor = (color, rating) => ({
    ...color,
    rating
})
```

exactly the same version as above but with cleaner syntax. Notice that we wrap the returned object in paraentheses. With arrow functions, this is required since the arrrow can't point to an object's curly baces.

Object modification can be done declaratively leveraging the function `javascript Object.assign({}, ...)` so, what about other data structure like array? in Javascript, don't use array method like `javascript Array.push(...)` to modify an array declaratively, since the method mutates the the array object on which it is called. instead, use method `javascript concat(...)` that create a copy of the array object on which it is called before modifying it.

```
/// Don't do this in declarative programming
let list = [
    {title: "Rad Red"},
    {title: "Lawn"},
    {title: "Party Pink"}
]

var addColor = function(title, colors) {
    colors.push({title: title})
    return colors
}
console.log(addColor("Gram Green", list).length) /// prints 4
console.log(list.length) /// prints 4
/// Instead, implement the following immutable function
/// this implementation takes a new object and adds it to a copy of the original
array
const addColor = (title, colors) => colors.concat({title})
console.log(addColor("Gram Green", list).length) /// prints 4
console.log(list.length) /// prints 3
```