Enumeration in JavaScript

For loops

JavaScript has for loops we can use for enumerating the elements of an array.

```
const colors = ['red', 'green', 'blue']

function logSomeColor(color: string, index: number) {
  console.log(`The color at position ${index} is ${color}`)
}

colors.forEach(logSomeColor)
```

Transforming values

- Instead of `console.log` let's build a *new array*
- Each element of this new array to be the equal to the **length** of the string at the corresponding index of the original array.

```
[                   [
  'red',      =>      3,
  'green',    =>      5,
  'blue'      =>      4,
]                   ]
```

```
const colors = ['red', 'green', 'blue']

// Code here

const lengths = [3, 5, 4]
```

Manually

We will start by doing this in a very manual way.

Begin by creating a new array to receive the individual elements.

```
const colors = ['red', 'green', 'blue']

const lengths: number[] = []
```

Then we will setup the forEach loop

```
const colors = ['red', 'green', 'blue']

const lengths: number[] = []

colors.forEach(function (color) {
  // Code here
})
```

Code inside the loop

```
const colors = ['red', 'green', 'blue']

const lengths: number[] = []

colors.forEach(function (color) {
  const lengthOfColor = color.length

  lengths.push(lengthOfColor)
})

console.log(lengths) // [ 3, 5, 4 ]
```

This is a fairly simple loop with a few steps within the loop itself.

Issues

- It does not allow us to use it in a generic way.
- Another transformation (upper-casing) would require another copy of the loop.

```
const colors = ['red', 'green', 'blue']

const lengths: number[] = []

colors.forEach(function (color) {
  const lengthOfColor = color.length

  lengths.push(lengthOfColor)
})

console.log(lengths) // [ 3, 5, 4 ]
```

```
const uppercased: string[] = []

colors.forEach(function (color) {
  const uppercase = color.toUpperCase()

  uppercased.push(uppercase)
})

console.log(uppercased) // [ 'RED', 'GREEN', 'BLUE' ]
```

Introduce `map`

```
const colors = ['red', 'green', 'blue']

const lengths = colors.map(function (color) {
  const lengthOfColor = color.length

  return lengthOfColor
})

console.log(lengths) // [ 3, 5, 4 ]
```

```
const uppercased = colors.map( function (color) {
  const uppercase = color.toUpperCase()

  return uppercase
})

console.log(uppercased) // [ 'RED', 'GREEN', 'BLUE' ]
```

Improvements

- We no longer have to initialize an empty array and modify its contents.
- We no longer push to the array, but simply **return** the new value from our callback function.

Refactor

We can simplify the code a little if we remove the temporary variables.

```
const colors = ['red', 'green', 'blue']

const lengths = colors.map(function (color) {
  return color.length
})

console.log(lengths) // [ 3, 5, 4 ]
```

Refactor

We can also use arrow `functions`

```
const colors = ['red', 'green', 'blue']

const lengths = colors.map(color => color.length)

console.log(lengths) // [ 3, 5, 4 ]

const uppercased = colors.map(color => color.toUpperCase())

console.log(uppercased) // [ 'RED', 'GREEN', 'BLUE' ]
```

Similar to LINQ

`map` is much like `Select`

filter

If we wish to create a new array but only retain *some* of the elements from the original array we can use `filter`

```
const colors = ['red', 'green', 'blue']

const longColors = colors.filter(color => color.length > 3)

console.log(longColors) // [ 'green', 'blue' ]
```

This is very similar to `C#` `Where` from LINQ.

reduce

```
const numbers = [100, 42, 13]

const total = numbers.reduce((total, number) => total + number, 0)

console.log(total) // [ 155 ]
```

Others

See the quick reference guide for other iterators such as `some`, `every`, and `reduce-right`.