WEP – Functions

Objects as associative arrays or dictionaries

Agenda

- Functions
- Functions Types
- Functions Expressions in Array

functions

Functions

- Functions are the main “building blocks” of the program.
- They allow the code to be called many times without repetition.

Function Types

- Function declaration
- Function expression

```
const count = function(array) {
 return array.length;
}
```

- Shorthand method definition

```
items: [],
add(...items)
```

- Arrow function

```
let sum = (a, b) => a + b;
```

- Generator function

Function Declaration

- The syntax that we used before is called a Function Declaration:

```
function sayHi() {
  alert( "Hello" );
}
```

Function Expression

- There is another syntax for creating a function that is called a Function Expression.
- It allows to create a new function in the middle of any expression.

```
let sayHi = function() {
  alert( "Hello" );
};
```

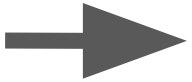
Arrow Function

- There's another very simple and concise syntax for creating functions, that's often better than Function Expressions.
- It's called "arrow functions", because it looks like this:

```
let sum = (a, b) => a + b;
```

Let's Try

```
function factorial(n) {
 if (n === 0) {
   return 1;
 }
 return n * factorial(n - 1);
}
```



Arrow Function

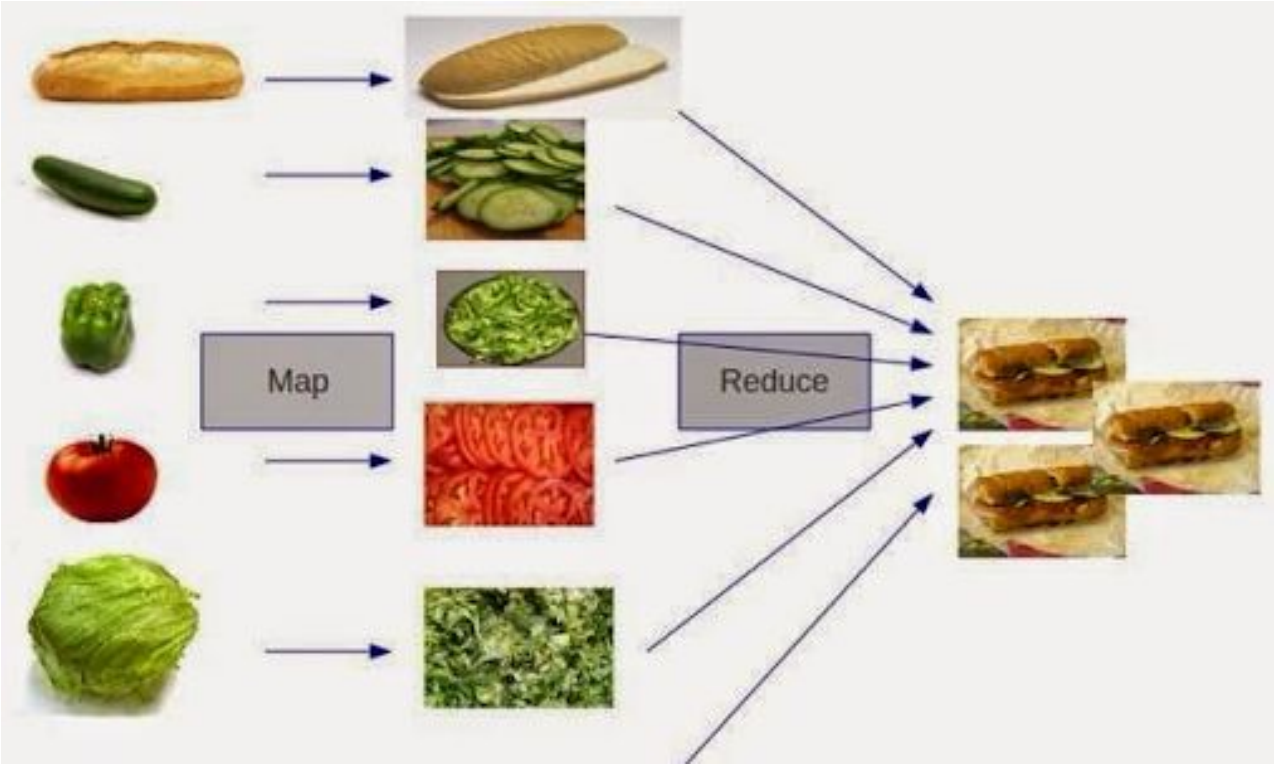
Functional extensions in arrays

eliminate the need of for-loops

Functional Extensions

Mostly used:

- Array.prototype.find()
- Array.prototype.filter()
- Array.prototype.map()
- Array.prototype.reduce()
- Array.prototype.every()
- Array.prototype.some()


Map
Reduce

map, filter, and reduce explained with emoji 😂

```
map([🐮, 🍆, 🐔, 🌽], cook)
=> [🍔, 🍟, 🍗, 🍿]

filter([🍔, 🍟, 🍗, 🍿], isVegetarian)
=> [🍟, 🍿]

reduce([🍔, 🍟, 🍗, 🍿], eat)
=> 💩
```

map, filter, reduce

Explained With Emoji 😂

```
let cooked = [🐮, 🌮, 🐓, 🌽, 🐟, 🐖, 🍋, 🍓]
    .map(cook) // [🍔, 🍟, 🍗, 🍿, 🥩, 🌭, 🍸, 🍨]

let vegetarian = [🍔, 🍟, 🍗, 🍿, 🥩, 🌭, 🍸, 🍨]
    .filter(isVegetarian) // [🍟, 🍿, 🍸, 🍨]

let reduction = [🍔, 🍟, 🍗, 🍿, 🥩, 🌭, 🍸, 🍨]
    .reduce(😪, eat) // "😋"
```

```
const inventory = [
    {name: 'apples', quantity: 2},
    {name: 'bananas', quantity: 0},
    {name: 'cherries', quantity: 5}
];

const result = inventory.find( fruit => fruit.name === 'cherries' );

console.log(result) // { name: 'cherries', quantity: 5 }
```

find

```
function isBigEnough(value) {
  return value >= 10;
}

var filtered = [12, 5, 8, 130, 44].filter(isBigEnough);
// filtered is [12, 130, 44]
```

filter

```
var kvArray = [{key: 1, value: 10},
               {key: 2, value: 20},
               {key: 3, value: 30}];

var reformattedArray = kvArray.map(obj =>{
  var rObj = {};
  rObj[obj.key] = obj.value;
  return rObj;
});
// reformattedArray is now [{1: 10}, {2: 20}, {3: 30}],

// kvArray is still:
// [{key: 1, value: 10},
//  {key: 2, value: 20},
//  {key: 3, value: 30}]
```

map

```
[0, 1, 2, 3, 4].reduce(function(accumulator, currentValue, currentIndex, array) {
  return accumulator + currentValue;
});
```

reduce

```
function isBiggerThan10(element, index, array) {
  return element > 10;
}

[2, 5, 8, 1, 4].some(isBiggerThan10);  // false
[12, 5, 8, 1, 4].some(isBiggerThan10); // true
```

some

```
function isBigEnough(element, index, array) {
  return element >= 10;
}
[12, 5, 8, 130, 44].every(isBigEnough);   // false
[12, 54, 18, 130, 44].every(isBigEnough); // true
```

every

```
var arr1 = [1, 2, [3, 4]];
arr1.flat();
// [1, 2, 3, 4]

var arr2 = [1, 2, [3, 4, [5, 6]]];
arr2.flat();
// [1, 2, 3, 4, [5, 6]]

var arr3 = [1, 2, [3, 4, [5, 6]]];
arr3.flat(2);
// [1, 2, 3, 4, 5, 6]
```

flat

```
var arr1 = [1, 2, 3, 4];

arr1.map(x => [x * 2]);
// [[2], [4], [6], [8]]

arr1.flatMap(x => [x * 2]);
// [2, 4, 6, 8]

// only one level is flattened
arr1.flatMap(x => [[x * 2]]);
// [[2], [4], [6], [8]]
```

flatmap

Let's Try

input	=>	let numberList = [2,3,4,5];
output	=>	let result = [[4],[9],[16],[25]];

Questions?