Functional Programming in JavaScript

MAP

Example:

```
var array1 = [1, 4, 9, 16];
const map1 = array1.map(x => x * 2);
// expected output: Array [2, 8, 18, 32]
```

```
var new_array = arr.map(function callback(currentValue[, index[, array]]) {
    // Return element for new_array
}[, thisArg])
```

Parameters

`callback`

Function that produces an element of the new Array, taking three arguments:

`currentValue`

The current element being processed in the array.

`index` Optional

The index of the current element being processed in the array.

`array` Optional

The array `map` was called upon.

`thisArg` Optional

Value to use as `this` when executing `callback`.

Return value

A new array with each element being the result of the callback function.

FILTER:

Example:

```
var words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];
const result = words.filter(word => word.length > 6);
// expected output: Array ["exuberant", "destruction", "present"]
```

```
var newArray = arr.filter(callback(element[, index[, array]])[, thisArg])
```

Parameters

`callback`

Function is a predicate, to test each element of the array. Return `true` to keep the element, `false` otherwise. It accepts three arguments:

`element`

The current element being processed in the array.

`index` Optional

The index of the current element being processed in the array.

`array` Optional

The array `filter` was called upon.

`thisArg` Optional

Optional. Value to use as `this` when executing `callback`.

Return value

A new array with the elements that pass the test. If no elements pass the test, an empty array will be returned.

REDUCE:

Example:

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;
console.log(array1.reduce(reducer));
// expected output: 10
```

> *arr*.reduce(*callback[, initialValue]*)

Parameters

`callback`

Function to execute on each element in the array, taking four arguments:

`accumulator`

The accumulator accumulates the callback's return values; it is the accumulated value previously returned in the last invocation of the callback, or `initialValue`, if supplied (see below).

`currentValue`

The current element being processed in the array.

`currentIndex` Optional

The index of the current element being processed in the array. Starts at index 0, if an `initialValue` is provided, and at index 1 otherwise.

`array` Optional

The array `reduce()` was called upon.

`initialValue` Optional

Value to use as the first argument to the first call of the `callback`. If no initial value is supplied, the first element in the array will be used. Calling `reduce()` on an empty array without an initial value is an error.

Return value

The value that results from the reduction.