<WA/>
2026

JS Callbacks and Functional Programming

“The” language of the Web

Fulvio Corno
Luigi De Russis
Enrico Masala

Outline

- Callbacks
- Functional Programming

JavaScript: The Definitive Guide, 7th Edition
11.1 Asynchronous Programming with Callbacks

JavaScript – The language of the Web

CALLBACKS

https://developer.mozilla.org/en-US/docs/Glossary/Callback_function

Callbacks

- A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.
  - Synchronous
  - Asynchronous

```
function logQuote(quote) {
  console.log(quote);
}

function createQuote(quote, callback) {
  const myQuote =
    `Like I always say, '${quote}'`;
  callback(myQuote);
}

createQuote("WebApp I rocks!", logQuote);
```

Synchronous Callbacks

- Used in functional programming
  - e.g., providing the sort criteria for array sorting
    NB: default sort order is ascending, with [number to string conversion (!)]

```
let numbers = [4, 2, 5, 1, 3];

numbers.sort(function(a, b) {
  return a - b;
});

console.log(numbers);
```

```
let numbers = [4, 2, 5, 1, 3];

numbers.sort((a, b) => a - b);

console.log(numbers);
```

Synchronous Callbacks

- Example: filter according to a criteria
  - filter() creates a **new** array with all elements for which the callback returns true

```
const market = [
  { name: 'GOOG', var: -3.2 },
  { name: 'AMZN', var:  2.2 },
  { name: 'MSFT', var: -1.8 }
];

const bad = market.filter(stock => stock.var < 0);
// [ { name: 'GOOG', var: -3.2 }, { name: 'MSFT', var: -1.8 } ]

const good = market.filter(stock => stock.var > 0);
// [ { name: 'AMZN', var: 2.2 } ]
```

JavaScript: The Definitive Guide, 7th Edition
Chapter 6. Array
Chapter 7.8 Functional Programming

JavaScript – The language of the Web

FUNCTIONAL PROGRAMMING

Functional Programming: A Brief Overview

- A programming paradigm where the developer mostly construct and structure code using *functions*
  - not JavaScript's main paradigm, but JavaScript is well suited
- More “declarative style” rather than “imperative style” (e.g., for loops)
- Can improve program readability:

```
new_array =
  array.filter ( filter_function ) ;
```

```
new_array = [] ;
for (const el of list)
       if ( filter_function(el) )
              new_array.push(el) ;
```

Notable Features of the Functional Paradigm

- Functions are *first-class* citizens
  - functions can be used as if they were variables or constants, combined with other functions and generate new functions in the process, chained with other functions, etc.
- *Higher-order functions*
  - a function that operates on functions, taking one or more functions as arguments and typically returning a new function
- Function *composition*
  - composing/creating functions to simplify and compress your functions by taking functions as an argument and return an output
- Call *chaining*
  - returning a result of the same type of the argument, so that multiple functional operators may be applied consecutively

Functional Programming in JavaScript

- JavaScript supports the features of the paradigm “out of the box”
- Functional programming requires *avoiding mutability*
  - i.e., do not change objects in place!
  - e.g., if you need to perform a change in an array, return a new array

Iterating over Arrays

Functional style

- Iterators: `for ... of`, `for (..;..;..)`
- Iterators: `forEach(f)`
  - Process each element with callback f
- Iterators: `every(f)`, `some(f)`
  - Check whether all/some elements in the array satisfy the Boolean callback `f`
- Iterators that return a new array: `map(f)`, `filter(f)`
  - Construct a new array
- `reduce`: callback function on all items to *progressively* compute a result

```
reduce(callback( accumulator, currentValue[, index[, array]] )[, initialValue])
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach

.forEach()

- `forEach()` invokes your (synchronous) callback function once for each element of an **iterable**

```
const letters = [..."Hello world"] ;
let uppercase = "" ;
letters.forEach(letter => {
  uppercase += letter.toUpperCase();
});
console.log(uppercase); // HELLO WORLD
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach

.forEach()

- `forEach()` invokes your (synchronous) callback function once for each element of an **iterable**
  - The callback may have 3 parameters
    - `currentValue`: The current element being processed in the array.
    - `index` (Optional): The index of `currentValue` in the array
    - `array` (Optional): The array forEach() was called upon.
  - Always **returns *undefined*** and is **<u>not</u> chainable**
  - No way to stop or break a `forEach()` loop other than by throwing an exception
- `forEach()` does not mutate the array on which it is called
  - however, its callback *may* do so

.every()

- `every()` tests whether **all elements** in the array pass the test implemented by the provided function
  - Callback: Same 3 arguments as forEach
  - It returns a Boolean value (*truthy/falsy*)
  - It executes its callback once for each element present in the array until it finds the one where the callback returns a falsy value
    - If such an element is found, **immediately** returns false

```
let a = [1, 2, 3, 4, 5];
a.every(x => x < 10); // => true: all values are < 10
a.every(x => x % 2 === 0); // false: not all even values
```

.some()

- `some()` tests whether **at least one** element in the array passes the test implemented by the provided function
  - It returns a Boolean value
  - It executes its callback once for each element present in the array until it finds the one where the callback returns a truthy value
    - if such an element is found, **immediately** returns true

```
let a = [1, 2, 3, 4, 5];
a.some(x => x%2===0); // => true; a has some even numbers
a.some(isNaN);
```

.map()

- `map()` passes each element of the **array** on which it is invoked to the function you specify
  - the callback should return a value
  - `map()` always returns **a *new* array** containing the values returned by the callback

```
const a = [1, 2, 3];

const b = a.map(x => x*x);

console.log(b); // [1, 4, 9]
```

```
const letters = [..."Hello world"];

const uppercase = letters.map(letter
=> letter.toUpperCase());

console.log(uppercase.join(''));
```

.filter()

- `filter()` creates **a *new* array** with all elements that pass the test implemented by the provided function
  - the callback is a function that returns either true or false
  - if no element passes the test, an empty array is returned

```
const a = [5, 4, 3, 2, 1];

a.filter(x => x < 3); // generates [2, 1], values less than 3

a.filter((element, index) => index%2 == 0); // [5, 3, 1]
```

.reduce()

```
reduce(
        callback(accumulator, currentValue[, index[, array]])
        [, initialValue]
)
```

- `reduce()` combines the elements of an **array**, using the specified function, to produce **a *single* value**
  - this is a common operation in functional programming and goes by the names “inject” and “fold”
- `reduce` takes two arguments:
  1. the *“reducer function”* (`callback`) that performs the reduction/combination operation (combine or **reduce 2 values into 1**)
  2. an (optional) **`initialValue`** to pass to the function; if not specified, it uses the first element of the array as initial value (and iteration starts from the next element)

.reduce()

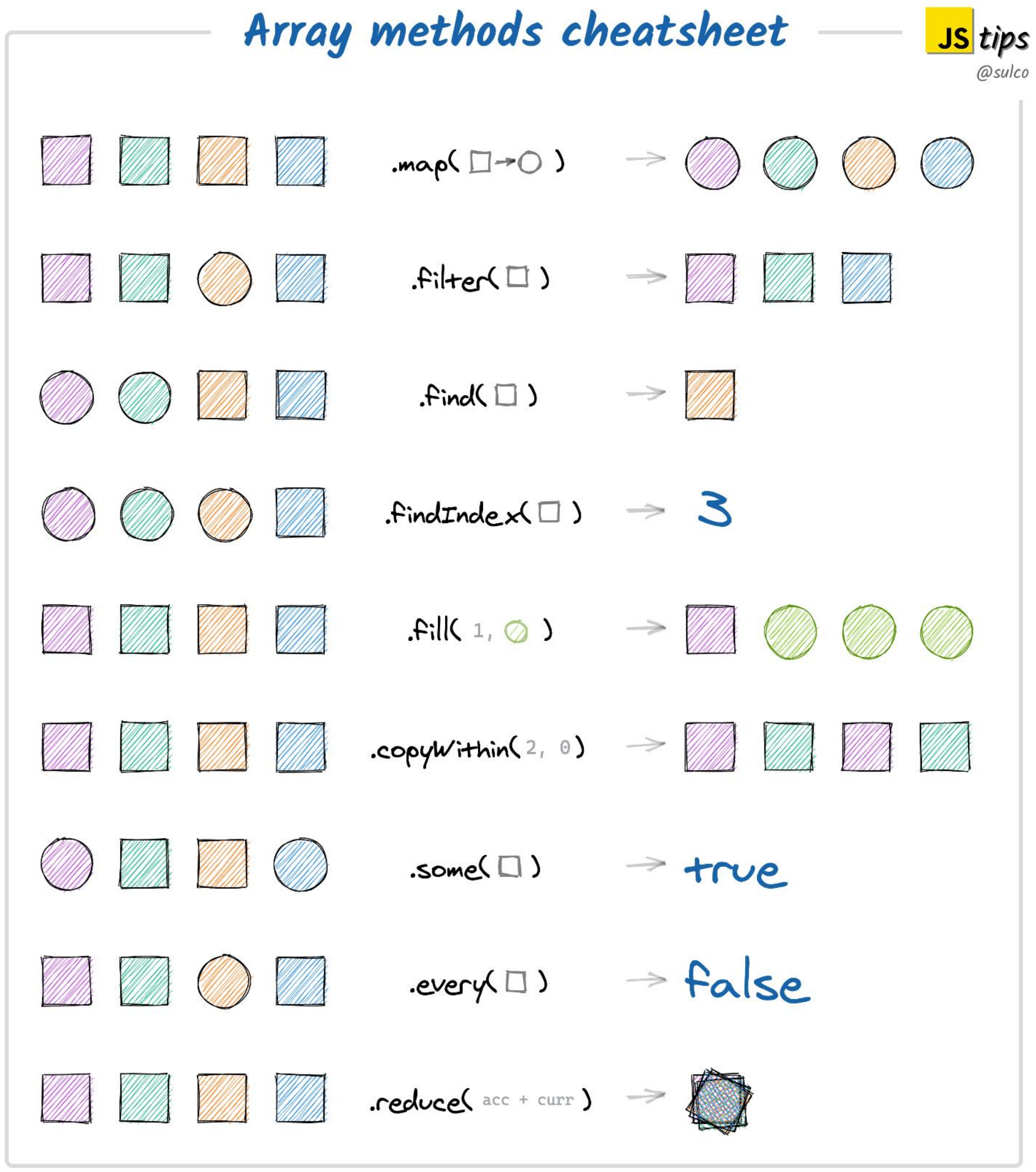
- Callbacks used with `reduce()` are different than the ones used with `forEach()` and `map()`
  - the *first* argument is the **accumulated result** of the reduction so far
  - on the first call to this function, its first argument is the initial value
  - on subsequent calls, it is the value returned by the previous invocation of the reducer function

```
const a = [5, 4, 3, 2, 1];

a.reduce( (accumulator, currentValue) =>
accumulator + currentValue,     0);
// 15; the sum of the values


a.reduce((acc, val) => acc*val,  1);
// 120; the product of the values


a.reduce((acc, val) => (acc > val) ? acc
: val);
// 5; the largest of the values
```


Array methods cheatsheet
JS tips
@sulco
.map(□→○)
.filter(□)
.find(□)
.findIndex(□)
3
.fill(1, ○)
.copyWithin(2, 0)
.some(□)
true
.every(□)
false
.reduce(acc + curr)

Example: average price of all SUVs

```
const vehicles = [
  { make: 'Honda',  model: 'CR-V',     type: 'suv',   price: 24045 },
  { make: 'Honda',  model: 'Accord',   type: 'sedan', price: 22455 },
  { make: 'Mazda',  model: 'Mazda 6',  type: 'sedan', price: 24195 },
  { make: 'Mazda',  model: 'CX-9',     type: 'suv',   price: 31520 },
  { make: 'Toyota', model: '4Runner',  type: 'suv',   price: 34210 },
  { make: 'Toyota', model: 'Sequoia',  type: 'suv',   price: 45560 },
  { make: 'Toyota', model: 'Tacoma',   type: 'truck', price: 24320 },
  { make: 'Ford',   model: 'F-150',    type: 'truck', price: 27110 },
  { make: 'Ford',   model: 'Fusion',   type: 'sedan', price: 22120 },
  { make: 'Ford',   model: 'Explorer', type: 'suv',   price: 31660 }
];

const averageSUVPrice = vehicles
  .filter(v => v.type === 'suv')
  .map(v => v.price)
  .reduce( (sum, price, i, array) => sum + price / array.length, 0);

console.log(averageSUVPrice); // 33399
```

https://opensource.com/article/17/6/functional-javascript

License