|  |  |
| --- | --- |
| **Functional Programming in JS: map, filter, reduce (Pt. 5)**  *Note: This is part of the “Javascript and Functional Programming” series on learning functional programming techniques in JavaScript ES6+. Checkout the previous post on function currying*[*<Part 4>*](https://hackernoon.com/javascript-and-functional-programming-currying-pt-4-96e3230782ab)*. Start from the*[*beginning here*](https://hackernoon.com/javascript-and-functional-programming-an-introduction-286aa625e26d)*.*  Let’s get practical! Previously, we examined *higher order functions.*As a quick reminder*,* a higher order function*is a***function that accepts another function as a parameter.**  Javascript arrays have several **built-in methods** that are higher order functions.  https://cdn-images-1.medium.com/max/800/1*FRAwpLpaqSMBV6VOHY_Vdw.png  This post will discuss the 3 most popular ones: *filter, map and reduce.*🙂🙂  **filter**  *the filter array method creates a new array with all elements that pass the test implemented by the provided function.*  That was straight [out of the docs](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter). In a friendlier way, filter is a method that runs on a given collection/array, and filters item based on a function that returns a boolean (true or false) value.  Let’s jump into an example and then step through what is actually happening. We will assume the following collection for our example.  const iceCreams = [  { flavor: 'pineapple', color: 'white' },  { flavor: 'strawberry', color: 'red' },  { flavor: 'watermelon', color: 'red' },  { flavor: 'kiwi', color: 'green' },  { flavor: 'mango', color: 'yellow' },  { flavor: 'pear', color: 'green' } ];  Let’s use the filter method to create a new array with *only red colored ice cream*. Remember, filter creates a new array, therefore we are going to have to save the output of the function into a variable to console it later.  const favoriteFlavors = iceCreams  .filter(iceCream => iceCream.color === 'red');  console.log(favoriteFlavors);  Running this snippet in the console will result in the following output:  https://cdn-images-1.medium.com/max/800/1*4YWAhyJYbdFDYtJTr57G0Q.png  If you’re feeling a little confused that’s fine. Let’s walk through it :)  https://cdn-images-1.medium.com/max/800/1*Mo0oHuz4hx4P8pmcXI5wfg.gif  the .***filter*** method accepts a function with 4 arguments, in the following order   * **element** -the current element of the array * **index**- the current index of the array (optional) * **array**- a reference to the array itself (optional) * **thisArg**-value to use as this when executing the callback (optional)   Altogether we need to supply a function with the following signature  https://cdn-images-1.medium.com/max/800/1*1_cXL8CA4cETkVtW5fbmhQ.png  Notice that in our example we passed an anonymous function (it is not named!) We could have selected to pass a named function as well, as so:  const getRed = icecream => icecream.color === 'red';  const favoriteFlavors = iceCreams  .filter(getRed);  console.log(favoriteFlavors);  The output will look like this:  https://cdn-images-1.medium.com/max/800/1*60EGsl46qLXtxxGZmq7a-g.png  Please note, that we passed the getRed function that is implicitly called with the element parameter.  *filter is a great way to quickly*parse data, leaving only what is relevant to us. The idea of passing a function that adheres to a certain structure may seem a bit awkward if your reading this for the first time, but ends up being quite powerful. First of all, this structure make it easier for developers to read each others code. Second of all, we are going to use this same pattern right now, when examining the *.map* array method 😎😎😎😎😎  **map**  The **map()** method creates a new array with the results of calling a provided function on every element in the calling array. Essentially, the map method *creates a new array*, based on the initial array. A quick look at the ***map()***signature:  https://cdn-images-1.medium.com/max/800/1*BkcYRGvVCLfOBYqDP2SBXg.png  Just like ***filter()***, ***map()*** is another higher order function. Similar to the ***filter()*** method structure we must pass a function here as well. But instead of filtering items in the original array, we transform data.  🤔🤔🤔🤔🤔🤔🤔🤔🤔🤔🤔🤔  Let’s look at an example to clarify! We will use the array from the previous example.  const iceCreams = [  { flavor: 'pineapple', color: 'white' },  { flavor: 'strawberry', color: 'red' },  { flavor: 'watermelon', color: 'red' },  { flavor: 'kiwi', color: 'green' },  { flavor: 'mango', color: 'yellow' },  { flavor: 'pear', color: 'green' } ];  Let’s suppose we want to create a new array of strings, with all the flavors of ice cream. Before we use ***map()***let’s try doingit the old school way with a classic for loop.  let flavors = []; for (let i = 0; i < iceCreams.length; i++) {  flavors.push(iceCreams[i].flavor) }  console.log(flavors);  https://cdn-images-1.medium.com/max/800/1*65Ly7CXSyJw9OokiQUfUPw.png  Our output from the for loop code snippet  Technically, this works well for this trivial example. But do you see some of the problems that creep up when using *for*loops like this? My pessimism sees 3 opportunities to mess up here 😳😳😳   * Defining the iterator value   let i = 0;   * Defining the end value for the *for*loop   i < iceCreams.length   * Incrementing the iterator value   i++  In different scenarios and depending on our data it is easy to write mistakes here. It could be a typo as simple as forgetting a semi colon, or mistakenly instantiation the iterator to the wrong value.  Let’s try the same thing with the ***map()*** method.  const iceCreams = [  { flavor: 'pineapple', color: 'white' },  { flavor: 'strawberry', color: 'red' },  { flavor: 'watermelon', color: 'red' },  { flavor: 'kiwi', color: 'green' },  { flavor: 'mango', color: 'yellow' },  { flavor: 'pear', color: 'green' } ];  const flavors = iceCreams.map(icecream => icecream.flavor) console.log(flavors)  https://cdn-images-1.medium.com/max/800/1*sANCYWy7AE5Qd8AHyy-W0A.png  OMG?!!  Our output is identical, but notice how clean and terse the code is. No indexes, no semi colons, no declaring data length! Since programming with ***map()*** and ***filter(),***I’ve noticed that the ease of using these functional methods grows significantly in comparison to loops, based on the complexity of our collections /arrays. These are definitely worth integrating into your day to day programming workflow.  **reduce**  Finally! As [Christian Sakai](https://medium.com/@christianmsakai) mentioned in a previous comment, reduce is the granddad / grandma of all of these methods 👵🏻👵🏻👵🏻👵🏻👵🏻👵🏻👵🏻  According to the [documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce):  *The***reduce()***method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.*  This is pretty cryptic! Let’s “reduce” the meaning of this to something simple. Let’s circle back to the ***filter()***, and ***map()*** methods. What do they have in common? Essentially, they transform a collection / array into a different collection / array. But these methods are specific in how they transform the data. In comparison, ***reduce()***is like the swiss army knife of list transformations. It can be used to express **any** transformation! In fact, we can even use ***reduce()*** to implement ***map()*** and ***filter().***Enough talking! Let’s take a look at the classic reduce example of summing up an array 🙃 🙃 🙃 🙃  **First with *for* loop**  const arr = [10,20,30]  let total = 0; for(let i = 0; i < arr.length; i++) {  total += arr[i]  }  console.log(total);  https://cdn-images-1.medium.com/max/800/1*mqcMkXc1McLHfMZVEtQHNQ.png  **Now with reduce**  const arr = [10, 20, 30];  const reducerFunction = (acc, currentItem) => acc + currentItem;  const sum = arr.reduce(reducerFunction, 0); console.log(sum);  https://cdn-images-1.medium.com/max/800/1*QsPPjyvJBtfdfy0A3oFqQA.png  It is clear that we got the same result. Let’s break it down. The built-in array method ***reduce()*****requires a callback function as the first parameter.**This callback function is predetermined in its input, accepting up to 4 arguments, similar to the callbacks that the ***filter() and map()***expect. Let’s look at the expected ***reducer()*** function signature.  https://cdn-images-1.medium.com/max/800/1*mQrjP0aMK7xY1_SSHO2kjg.png  ***reduce()***second parameter is optional and is the initialValue. When we iterate over an array and attempt to reduce it to a single value, it is recommended to instantiate the initial value. In our *array summation*example, we instantiated the *initialValue*to zero. What would happen had we not instantiated the initial value?  const arr = [10, 20, 30];  const reducerFunction = (acc, currentItem) => acc + currentItem;  // Not instantiating the initial value! const sum = arr.reduce(reducerFunction); console.log(sum);  https://cdn-images-1.medium.com/max/800/1*h6M04mrhUSiYo03X5Ei_xA.png  We get the same value! Why is that? According to the documentation, when no *initialValue*is supplied, the first element in the array will be used. Although this example works out without supplying an initial value, **I recommend getting used to always supplying one**. This is going to prevent future bugs, and also require you to think if the *reduction* you’re trying to do on your array makes sense.  **Implementing map() and filter() with reduce()**  Earlier I claimed that ***reduce()***is the grandfather of list transformation methods, because we can use it to implement all of them. Let’s prove this!  ***map with reduce()***  // \*\*\*\*\*\*\*\*\*\*\*\*\* Map with Reduce \*\*\*\*\*\*\*\*\*\*\*\*\*  const data = [10, 20, 30];  const tripledWithMap = data.map(item => {  return item \* 3; });  const tripledWithReduce = data.reduce((acc, value) => {  acc.push(value \* 3);  return acc; }, []);  console.log(tripledWithMap, tripledWithReduce);  https://cdn-images-1.medium.com/max/800/1*7FKiyTWLR3yhedGfFJ7VEA.png  ***filter with reduce()***  // \*\*\*\*\*\*\*\*\*\*\*\*\* Filter with Reduce \*\*\*\*\*\*\*\*\*\*\*\*\*  const data2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  const evenWithFilter = data2.filter(item => {  return item % 2 === 0; })  const evenWithReduce = data2.reduce((acc, value) => {  if (value % 2 === 0) {  acc.push(value);  }  return acc; }, []);  console.log(evenWithFilter, evenWithReduce);  https://cdn-images-1.medium.com/max/800/1*J01FkBbvZIz8pwf_AZhckQ.png  **A more practical reduce() example**  Let’s do something a bit more useful with our ***reduce()***method. Let’s calculate the results of a vote for the best ice cream flavor 🍦🍦🍦🍦🍓🍋 🍌 🍉 🍇  const flavours = [  "strawberry",  "strawberry",  "kiwi",  "kiwi",  "kiwi",  "strawberry",  "mango",  "kiwi",  "banana" ];  const votes = {}; const reducer = (votes, vote) => {  votes[vote] = !votes[vote] ? (votes[vote] = 1) : votes[vote] + 1;  return votes; }; const outcome = flavours.reduce(reducer, votes);  // Output console.log("Strawberry: ", outcome.strawberry); console.log("Kiwi: ", outcome.kiwi); console.log("Mango: ", outcome.mango); console.log("Banana: ", outcome.banana);  https://cdn-images-1.medium.com/max/800/1*OBSzBJKAvoKvXKXa1OxJBw.png  Similar to any time we will use reduce, we are calling the ***reduce()*** method on an array and supplying a callback and initialValue. Notice that in this example, we set the initial value to an empty object. Without that initialization this would have failed!  **Flattening data with reduce**  First of all, let’s define flattening data for our example. Flattening looks like this:  [[a, b, c], [d, e, f], [g, h i]] -> [a, b, c, d, e, f, g, h, i]  Essentially we want to merge all the arrays in the order in which they appear. ***reduce()*** solves this elegantly 🤗🤗  const letterArr = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]; const flattened = letterArr.reduce((acc, val) => {  return acc.concat(val); }, []);  console.log(flattened);  https://cdn-images-1.medium.com/max/800/1*vs11Ye6h4uYUXMwmW-_Zng.png  **Performance with list transformers**  A common pattern is to chain list transformers. Although it makes complicated transformations easier to read it lacks in speed when dealing with very large arrays. Let’s look at an example.  let bigData = []; for (let i = 0; i < 1000000; i++) {  bigData[i] = i; }  // Slow let filterBegin = Date.now(); const filterMappedBigData = bigData  .filter(value => value % 2 === 0)  .map(value => value \* 2);   let filterEnd = Date.now(); let filtertimeSpent = (filterEnd - filterBegin) / 1000 + "secs";  // Fast let reducedBegin = Date.now(); const reducedBigData = bigData.reduce((acc, value) => {  if (value % 2 === 0) {  acc.push(value \* 2);  }  return acc; }, []); let reducedEnd = Date.now(); let reducedtimeSpent = (reducedEnd - reducedBegin) / 1000 + " secs";  console.log("filtered Big Data:", filtertimeSpent); console.log("reduced Big Data:", reducedtimeSpent);  https://cdn-images-1.medium.com/max/800/1*jJ6M5zf8wGJcJF5K_iEFKA.png  Why is the chaining of filter and map so slow? First ***filter()*** needs to iterate through the whole array (1,000,000) and filter half. Then ***map()*** iterates through the array remnants (500,000) and creates new array. In comparison, in ***reduce()*** we only iterate through array once! Not constantly iterating over the same data set is more efficient but less performant. | **Функциональное программирование в JS: map, filter, reduce (ч.5)**  Давайте перейдем сразу к практике! До этого мы изучали *функции высшего порядка.* Для тех, кто подзабыл, – это **функции, параметром которых может выступать другая функция.**  В массивах Javascript существует несколько **встроенных методов**, представляющих собой функции высшего порядка.  https://cdn-images-1.medium.com/max/800/1*FRAwpLpaqSMBV6VOHY_Vdw.png  В данной статье мы рассмотрим 3 самых популярных метода: *filter, map и reduce.*  **filter**  *Метод filter (Array) создает новый массив со всеми элементами, соответствующими требованиям данной функции.*  Формулировка [взята из учебника](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter). Если говорить простым языком, то filter – это метод, выполняемый в данной коллекции/массиве. А элемент фильтра основан на функции, которая возвращает логическое значение (true или false).  Давайте перейдем к примеру и посмотрим, что там происходит. В качестве примера возьмем следующую коллекцию:  const iceCreams = [  { flavor: 'pineapple', color: 'white' },  { flavor: 'strawberry', color: 'red' },  { flavor: 'watermelon', color: 'red' },  { flavor: 'kiwi', color: 'green' },  { flavor: 'mango', color: 'yellow' },  { flavor: 'pear', color: 'green' } ];  Воспользуемся методом *filter* для создания нового массива *только* с *мороженым красного цвета*. Запомните: фильтр создает новый массив, поэтому необходимо сохранять выход функции в переменную для последующего доступа к ней через консоль.  const favoriteFlavors = iceCreams  .filter(iceCream => iceCream.color === 'red');  console.log(favoriteFlavors);  Запуск этой части кода в консоли выдаст следующий результат:  Запутались? Это совершенно нормально. Давайте вместе во всем разберемся :)  https://cdn-images-1.medium.com/max/800/1*Mo0oHuz4hx4P8pmcXI5wfg.gif  Метод .***filter*** принимает функцию с 4 аргументами, расположенными в следующем порядке:   * **element** – текущий элемент массива; * **index** – текущий индекс массива (необязательное значение); * **array** **–** ссылка на сам массив (необязательное значение); * **thisArg** – используемое значение при выполнении обратного вызова (необязательное значение).   Итого, функция представляется со следующей сигнатурой:  **Объяснение сигнатуры filter ()**  Обязательный аргумент (фиол.). Текущий элемент, который обрабатывает массив.  Необязательный аргумент (гол.). Индекс текущего элемента, который обрабатывает массив.  Необязательный аргумент (гол.). Вызов filter () в массиве.  \*\*\* обратный вызов, передаваемый в filter, должен возвращать логическое значение  Обратите внимание, что в нашем примере передавалась анонимная функция (не именованная!). Именованная передавалась бы вот так:  const getRed = icecream => icecream.color === 'red';  const favoriteFlavors = iceCreams  .filter(getRed);  console.log(favoriteFlavors);  Вывод такой функции:  Учтите, что мы передавали функцию getRed с неявным вызовом параметром элемента.  *filter отлично подходит для быстрого* разбора данных, оставляя только то, что нам действительно нужно. На первый взгляд, идея передачи функций с определенной структурой может показаться достаточно странной, но со временем вы разглядите в ней довольно мощный инструмент для реализации поставленных задач. Во-первых, такая структура облегчает процесс чтения как своего, так и чужого кода. Во-вторых, мы можем использовать этот шаблон уже сейчас, при разборе метода *.map (Array)*. 😎😎😎😎😎  **map**  Метод **map()** создает новый массив с результатами вызова представленной функции по каждому элементу вызываемого массива. Суть в том, что этот метод берет исходный массив и на основании него *создает новый.* Краткая сигнатура ***map()***:  **Объяснение сигнатуры map ()**  Обязательный аргумент (фиол.). Текущий элемент, который обрабатывает массив.  Необязательный аргумент (гол.). Индекс текущего элемента, который обрабатывает массив.  Необязательный аргумент (гол.). Вызов map () в массиве.  \*\*\* обратный вызов, передаваемый в map, должен возвращать значение для newArray  Оба метода – и  ***filter()***, и ***map()*** – это функции высшего порядка. В **map ()**  мы тоже передаем функцию. Но вместо сортировки исходного массива, мы выполняем преобразование данных.  🤔🤔🤔🤔🤔🤔🤔🤔🤔🤔🤔🤔  Давайте разберем все на примере! Нам потребуется массив из предыдущего примера.  const iceCreams = [  { flavor: 'pineapple', color: 'white' },  { flavor: 'strawberry', color: 'red' },  { flavor: 'watermelon', color: 'red' },  { flavor: 'kiwi', color: 'green' },  { flavor: 'mango', color: 'yellow' },  { flavor: 'pear', color: 'green' } ];  Предположим, нам нужно создать новый массив строк – в них задан вкус мороженого. До того, как обратиться к ***map(),***решим задачку по старинке – классическим циклом.  let flavors = []; for (let i = 0; i < iceCreams.length; i++) {  flavors.push(iceCreams[i].flavor) }  console.log(flavors);  https://cdn-images-1.medium.com/max/800/1*65Ly7CXSyJw9OokiQUfUPw.png  Выходные значения от выполнения участка кода с циклом  С технической точки зрения такие простые примеры реализуются неплохо. Но знаете ли вы о потенциальных проблемах, которые могут возникать при употреблении *for* в цикле? С моим пессимизмом я вижу сразу 3 недочета 😳😳😳   * Определение значения итератора   let i = 0;   * Определение конечного значения для цикла *for*   i < iceCreams.length   * Увеличение итератора   i++  При разных сценариях и данных тут можно допустить массу ошибок. Например, пропустить какую-то букву, забыть про точку с запятой или по ошибке задать итератора с некорректным значением.  Теперь давайте попробуем сделать тоже самое, но с методом ***map()***.  const iceCreams = [  { flavor: 'pineapple', color: 'white' },  { flavor: 'strawberry', color: 'red' },  { flavor: 'watermelon', color: 'red' },  { flavor: 'kiwi', color: 'green' },  { flavor: 'mango', color: 'yellow' },  { flavor: 'pear', color: 'green' } ];  const flavors = iceCreams.map(icecream => icecream.flavor) console.log(flavors)  https://cdn-images-1.medium.com/max/800/1*sANCYWy7AE5Qd8AHyy-W0A.png  ЧУДЕСА?!!  Результат такой же, но обратите внимание, какой чистый и лаконичный получился код. Никаких индексов, точек с запятой, объявления длины данных! Начав программировать с ***map()*** и ***filter(),***я заметил, что плюсы от использования этих методов продолжают расти, чего не скажешь о циклах, основанных на сложности коллекции / массива. ***map ()*** и ***filter ()*** гарантированно стоит внедрять в свою каждодневную работу.  **reduce**  Наконец-то! Как метко выразился [Кристиан Сакай](https://medium.com/@christianmsakai) в своем комментарии, *reduce* – это прародитель всех наших методов 👵🏻👵🏻👵🏻👵🏻👵🏻👵🏻👵🏻  [Официльное](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce) определение гласит:  *Метод***reduce()***применяет функцию к аккумулятору и каждому элементу массива (слева направо), сводя их к одному значению.*  **Объяснение сигнатуры reduce ()**  Обязательный аргумент (фиол.). Аккумулятор накапливает возвращенные значения обратного вызова. Это накопленное значение, которое уже возвращалось в предыдущем обращении к функции обратного вызова или initialValue (если задано, см. ниже).  Обязательный аргумент (фиол.). Текущий элемент, который обрабатывает массив.  Необязательный аргумент (гол.). Индекс текущего элемента, который обрабатывает массив. Начинается с индекса 0 (если присутствует initialValue) или 1 (при отсутствие такового).  Необязательный аргумент (гол.). Вызов reduce () в массиве  Звучит как-то малопонятно! Попробуем упростить формулировку для большего понимания. Давайте вернемся к методам ***filter()*** и ***map().*** Что у них общего? Все они преобразуют одну коллекцию / массив в другую. А разница в том, как именно происходит преобразование данных. Образно говоря, ***reduce() –*** швейцарский армейский нож в любом преобразовании списка. Он используется для **любого** преобразования! По сути, мы можем применять ***reduce()*** для реализации ***map()*** и ***filter().*** Хватит разговоров! Перейдем к классическому примеру работы с сокращением массива 🙃 🙃 🙃 🙃  **Первый пример с циклом *for***  const arr = [10,20,30]  let total = 0; for(let i = 0; i < arr.length; i++) {  total += arr[i]  }  console.log(total);  https://cdn-images-1.medium.com/max/800/1*mqcMkXc1McLHfMZVEtQHNQ.png  **А теперь с *reduce***  const arr = [10, 20, 30];  const reducerFunction = (acc, currentItem) => acc + currentItem;  const sum = arr.reduce(reducerFunction, 0); console.log(sum);  https://cdn-images-1.medium.com/max/800/1*QsPPjyvJBtfdfy0A3oFqQA.png  Само собой, результат у них одинаковый. Но разберем все в деталях. Встроенный метод массива ***reduce()*****требует****для своего первого параметра функцию обратного вызова.**Эта функция имеет заранее определенное входное значение, принимает 4 аргумента, похожих на обратные вызовы из ***filter()*** и ***map()***. Рассмотрим ожидаемую сигнатуру функции ***reducer()***.  **Объяснение сигнатуры reduce ()**  Обязательный аргумент (фиол.). Аккумулятор накапливает возвращенные значения обратного вызова. Это накопленное значение, которое уже возвращалось в предыдущем обращении к функции обратного вызова или initialValue (если задано, см. ниже).  Обязательный аргумент (фиол.). Текущий элемент, который обрабатывает массив.  Необязательный аргумент (гол.). Индекс текущего элемента, который обрабатывает массив. Начинается с индекса 0 (если присутствует initialValue) или 1 (при отсутствие такового).  Необязательный аргумент (гол.). Вызов reduce () в массиве  Второй параметр в ***reduce()***не обязателен. Им является initialValue. В случаях, когда мы перебираем элементы массива и пытаемся сократить его до одного значения, рекомендуют конкретизировать начальное значение. В нашем примере *суммирования массивов* мы определяли *initialValue*равным нулю. А что бы произошло, если бы мы не стали конкретизировать начальное значение?  const arr = [10, 20, 30];  const reducerFunction = (acc, currentItem) => acc + currentItem;  // Not instantiating the initial value! const sum = arr.reduce(reducerFunction); console.log(sum);  https://cdn-images-1.medium.com/max/800/1*h6M04mrhUSiYo03X5Ei_xA.png  Мы получим то же значение! Почему так? Спецификация метода говорит о том, что в случаях, когда не задано значение *initialValue*, используется первый элемент массива. Несмотря на то, что наш пример выполнился и без установленного начального значения, **я настоятельно рекомендую приучать себя к тому, что задавать начальное значение – нужно!** Такой подход поможет предотвратить возможные ошибки и заставит вас лишний раз призадуматься о целесообразности *сокращения* вашего массива.  **Реализация map() и filter() с reduce()**  Ранее я говорил о том, что ***reduce()***является прародителем методов преобразования списков, потому как его можно использовать для реализации их всех. Давайте докажем это на практике!  ***map с reduce()***  // \*\*\*\*\*\*\*\*\*\*\*\*\* Map with Reduce \*\*\*\*\*\*\*\*\*\*\*\*\*  const data = [10, 20, 30];  const tripledWithMap = data.map(item => {  return item \* 3; });  const tripledWithReduce = data.reduce((acc, value) => {  acc.push(value \* 3);  return acc; }, []);  console.log(tripledWithMap, tripledWithReduce);  https://cdn-images-1.medium.com/max/800/1*7FKiyTWLR3yhedGfFJ7VEA.png  ***filter с reduce()***  // \*\*\*\*\*\*\*\*\*\*\*\*\* Filter with Reduce \*\*\*\*\*\*\*\*\*\*\*\*\*  const data2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  const evenWithFilter = data2.filter(item => {  return item % 2 === 0; })  const evenWithReduce = data2.reduce((acc, value) => {  if (value % 2 === 0) {  acc.push(value);  }  return acc; }, []);  console.log(evenWithFilter, evenWithReduce);  https://cdn-images-1.medium.com/max/800/1*J01FkBbvZIz8pwf_AZhckQ.png  **Более практичный пример с reduce()**  Самое время придумать более полезный пример с методом ***reduce()***. Можно рассчитать результаты голосования за лучший вкус мороженого 🍦🍦🍦🍦🍓🍋 🍌 🍉 🍇  const flavours = [  "strawberry",  "strawberry",  "kiwi",  "kiwi",  "kiwi",  "strawberry",  "mango",  "kiwi",  "banana" ];  const votes = {}; const reducer = (votes, vote) => {  votes[vote] = !votes[vote] ? (votes[vote] = 1) : votes[vote] + 1;  return votes; }; const outcome = flavours.reduce(reducer, votes);  // Output console.log("Strawberry: ", outcome.strawberry); console.log("Kiwi: ", outcome.kiwi); console.log("Mango: ", outcome.mango); console.log("Banana: ", outcome.banana);  https://cdn-images-1.medium.com/max/800/1*OBSzBJKAvoKvXKXa1OxJBw.png  Каждый раз при операции сокращения необходимо вызвать в массиве метод ***reduce()***,а также обозначить обратный вызов и initialValue. Обратите внимание, что в данном примере мы задаем изначальное значение в виде пустого объекта. Без такого определения ничего не заработает!  **Сглаживание данных с reduce**  Для начала давайте определим сглаживаемые данные. Сглаживание выглядит так:  [[a, b, c], [d, e, f], [g, h i]] -> [a, b, c, d, e, f, g, h, i]  То есть мы хотим объединить все массивы по порядку их отображения. ***reduce()*** элегантно справляется с решением этой проблемы 🤗🤗  const letterArr = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]; const flattened = letterArr.reduce((acc, val) => {  return acc.concat(val); }, []);  console.log(flattened);  https://cdn-images-1.medium.com/max/800/1*vs11Ye6h4uYUXMwmW-_Zng.png  **Производительность преобразователей списков**  Создание цепочки преобразователей списка – явление довольно частое. Несмотря на то, что прочесть сложные преобразования будет легче, этот метод явно проиграет по скорости, особенно при работе с очень большими массивами. Взгляните на пример.  let bigData = []; for (let i = 0; i < 1000000; i++) {  bigData[i] = i; }  // Slow let filterBegin = Date.now(); const filterMappedBigData = bigData  .filter(value => value % 2 === 0)  .map(value => value \* 2);   let filterEnd = Date.now(); let filtertimeSpent = (filterEnd - filterBegin) / 1000 + "secs";  // Fast let reducedBegin = Date.now(); const reducedBigData = bigData.reduce((acc, value) => {  if (value % 2 === 0) {  acc.push(value \* 2);  }  return acc; }, []); let reducedEnd = Date.now(); let reducedtimeSpent = (reducedEnd - reducedBegin) / 1000 + " secs";  console.log("filtered Big Data:", filtertimeSpent); console.log("reduced Big Data:", reducedtimeSpent);  https://cdn-images-1.medium.com/max/800/1*jJ6M5zf8wGJcJF5K_iEFKA.png  Почему цепочка **filter** и **map** такая медленная? Сначала ***filter()*** должен выполнить итерацию всего массива (1,000,000) и отфильтровать половину. Затем уже ***map()*** проходит по оставшейся части массива (500,000) и создает новый массив. Для сравнения: в ***reduce()*** мы перебираем массив только раз! Отсутствие постоянной итерации одинакового набора данных более результативно в плане качества, но менее эффективно по части производительности. |

https://hackernoon.com/functional-programming-in-js-map-filter-reduce-pt-5-308a205fdd5f