



ПРОТОТИП МАССИВА, ФУНКЦИИ ВЫСШЕГО ПОРЯДКА



ЕВГЕНИЙ ШЕК



ЕВГЕНИЙ ШЕК

Frontend разработчик
Aviata.kz / Chocotravel.kz



shek.e.g@yandex.ru



[@usertel](https://t.me/usertel)

ПЛАН ЗАНЯТИЯ

1. Константы и массивы
2. Поиск в массиве
3. Функции высшего порядка
4. Расширенный поиск в массиве
5. Преобразование массива

КОНСТАНТЫ И МАССИВЫ

ВСПОМНИМ КОНСТАНТЫ

Вопрос: Что будет выведено в консоль при выполнении этого блока кода?

```
1 const age = 21;  
2 age++;  
3  
4 console.log(age);
```

ИЗМЕНЯТЬ КОНСТАНТЫ НЕЛЬЗЯ

Ответ: на консоль будет выведено сообщение об ошибке, полученной при попытке изменения значения

```
// TypeError: Assignment to constant variable.
```

Как вы помните изменение значения константы приводит к ошибке, о чем и написано в сообщении (Присваивание значения константе).

МЕТОДЫ ИЗМЕНЕНИЯ МАССИВА

Вопрос: А что если в константе массив, и мы пытаемся его изменить. Что будет выведено на консоль?

```
1 const numbers = [1, 2, 3];
2 numbers.shift();
3 numbers.pop();
4
5 console.log(numbers);
```

МАССИВ МЕНЯТЬ МОЖНО!

Ответ:

```
// [2]
```

Как видим `shift` и `pop` отработали и внесли изменения в массив.

Мы выяснили, что массив сохраненный в константе можно изменять с помощью его методов.

КОНСТАНТА ЗАПРЕЩАЕТ ТОЛЬКО ПРИСВАИВАНИЕ

Константы запрещают только присваивание в переменную, и любые другие операции, которые основаны на присваивании (`+=`, `-=`, `%=`, `++`, `--` и др.), даже если это массив:

```
1 const numbers = [1, 2, 3];
2 numbers = [4, 5, 6];
3 // Uncaught TypeError: Assignment to constant variable.
4 const [ firstNumber ] = numbers;
5
6 console.log(firstNumber);
```

МУТАБЕЛЬНОСТЬ И ИММУТАБИЛЬНОСТЬ

Объекты, имеющие методы, которые позволяют менять сам объект, или свойства, которые можно менять напрямую, являются изменяемыми, мутабельными (от слова *мутация, мутант*).

В JavaScript все объекты, в том числе и массивы – изменяемы, т.е. мутабельны (*mutable*).

Запомните еще раз, оператор `const` не делает объекты и массивы неизменными, иммутабельными. Он только ограничивает операции присваивания в переменную.

МАССИВ ЭТО ОБЪЕКТ

Кстати, массив это объект, в этом легко убедиться на этом примере кода:

```
console.log(typeof []); // object
```

ПРОВЕРКА НА МАССИВ

У нас есть функция, которая выводит в консоль содержимое массива. Что будет если в нее передать не массив?

```
1 function logArray(arr) {  
2     console.log(`Массив размером ${arr.length}`);  
3     for (let item of arr) {  
4         console.log(`${item} (${typeof item})`);  
5     }  
6 }  
7  
8 logArray(12);  
9 // Массив размером undefined  
10 // TypeError: arr is not iterable
```

Как мы можем проверить аргумент и защититься от неправильного использования функции?

ФУНКЦИЯ `Array.isArray` И ПЕРЕХВАТ ИСКЛЮЧЕНИЙ

Ранее мы уже выяснили, что `typeof` для массива возвращает `object`. Поэтому нужно другое решение. Для этого можно использовать функцию `Array.isArray`:

```
1 function logArray(arr) {
2     if (!Array.isArray(arr)) {
3         throw new Error('Аргумент не является массивом'); // выбрасываем исключение
4     }
5     console.log(`Массив размером ${arr.length}: [`);
6     for (let item of arr) {
7         console.log(`${item} (${typeof item})`);
8     }
9     console.log(`]`);
10}
11
12try {
13    logArray(12);
14} catch (e) { // отлавливаем исключение
15    console.log(e);
16}
17// Error: Аргумент не является массивом
```

ПРЕОБРАЗОВАНИЕ В МАССИВ СТРОК С ПОМОЩЬЮ `Array.from`

Функция `Array.from`, позволяет создавать массивы из итерируемых и массивоподобных объектов:

```
1 const str = 'Не массив!';
2
3 const symbols = Array.from(str);
4
5 console.log(symbols);
```

Итерируемый объект – объект, у которого есть итератор, изучите их в продвинутой версии курса. Массивоподобный объект – объект, у которого есть свойство `length` и числовые ключи.

ФУНКЦИЯ ОБЪЯВЛЕННАЯ БЕЗ АРГУМЕНТОВ

Допустим, вы создали функцию без аргументов:

```
1 | function logArgs() {  
2 |  
3 |}  
4 |  
5 | logArgs(1, 2, 3);
```

Как в теле функции обратиться к аргументам, переданным при вызове функции?

МАССИВОДОБНЫЙ ОБЪЕКТ `arguments`

Внутри каждой функции доступна переменная `arguments`, которая предоставляет альтернативный доступ ко всем переданным в функцию аргументам:

```
1 function logArgs() {  
2     console.log(arguments);  
3     // Arguments[1,2,3]  
4     console.log(arguments.length);  
5     // 3  
6     console.log(Array.isArray(arguments));  
7     // false  
8     console.log(arguments.join(' ', ' '));  
9     // Uncaught TypeError: arguments.join is not a function  
10 }  
11  
12 logArgs(1, 2, 3);
```

Мы видим, что `arguments` не является массивом, хотя и очень на него похож. И у него нет метода `join`.

ПРЕОБРАЗУЕМ `arguments` В МАССИВ

Вот тут нам и поможет функция `Array.from`, чтобы получить полноценный массив:

```
1 function logArgs() {  
2   console.log(Array.from(arguments).join(', '));  
3 }  
4  
5 logArgs(1, 2, 3); // 1, 2, 3
```

Мы видим, что `arguments` не является массивом, хотя и очень на него похож. И у него нет метода `join`.

ПОИСК В МАССИВЕ

ИНДЕКС ИСКОМОГО ЭЛЕМЕНТА

Допустим, у нас есть массив имен, и мы хотим выяснить, под каким индексом в массиве хранится имя `Иван`:

```
const names = [ 'Денис' , 'Егор' , 'Петр' , 'Иван' , 'Олег' ];
```

Вариант с перебором в цикле `for-of` не подходит, потому что там нет индекса. Остаются `for-in` или `for`.

Но есть способ лучше.

МЕТОД МАССИВА `indexOf`

У массива есть метод `indexOf`, который строго сравнивает аргумент с каждым элементом массива и возвращает индекс первого элемента, сравнение с которым вернет истину:

```
1 | const names = [ 'Денис', 'Егор', 'Петр', 'Иван', 'Олег' ];
2 |
3 | console.log(names.indexOf('Иван')); // 3
4 | console.log(names.indexOf('Егор')); // 1
5 | console.log(names.indexOf('Маша')); // -1
```

Обратите внимание, что возвращает индекс, а не номер. Индексы начинаются с 0. Метод `indexOf` вернет `-1`, если искомого элемента нет.

ЧАСТАЯ ОШИБКА С `indexOf`

Создадим функцию, которая добавляет элемент в массив только если его там еще нет:

```
1 function pushOnce(arr, item) {  
2     if (!arr.indexOf(item)) {  
3         arr.push(item);  
4     }  
5 }  
6  
7 const uniqNumbers = [];  
8 pushOnce(uniqNumbers, 1);  
9 pushOnce(uniqNumbers, 2);  
10 pushOnce(uniqNumbers, 2);  
11 pushOnce(uniqNumbers, 1);  
12  
13 console.log(uniqNumbers); // []
```

Массив пуст, хотя там должно быть два элемента. Почему?

ПРАВИЛЬНАЯ ПРОВЕРКА НАЛИЧИЯ

Когда элемент не найден, `index0f` возвращает `-1`, именно это и нужно проверять в функции `pushOnce`:

```
1 function pushOnce(arr, item) {  
2     if (arr.indexOf(item) === -1) {  
3         arr.push(item);  
4     }  
5 }  
6  
7 const uniqNumbers = [];  
8 pushOnce(uniqNumbers, 1);  
9 pushOnce(uniqNumbers, 2);  
10 pushOnce(uniqNumbers, 2);  
11 pushOnce(uniqNumbers, 1);  
12  
13 console.log(uniqNumbers); // [1,2]
```

Теперь функция работает, как задумывалось.

ЛУЧШАЯ ПРОВЕРКА НАЛИЧИЯ

Метод `indexOf` часто используется в условиях, и многие часто допускают ошибку проверки. Чтобы полностью исключить вероятность такой ошибки, используйте метод `includes`. Он идентичен методу `indexOf`, только вместо индекса он возвращает `true`, если искомый элемент есть в массиве, иначе возвращает `false`.

Перепишем функцию `pushOnce`, используя более подходящий метод:

```
1 function pushOnce(arr, item) {
2   if (!arr.includes(item)) {
3     arr.push(item);
4   }
5 }
6
7 let numbers = [];
8 pushOnce(numbers, 1);
9 pushOnce(numbers, 2);
10 pushOnce(numbers, 2);
11 pushOnce(numbers, 3);
12 console.log(numbers); // [1, 2, 3]
```

ЕСЛИ ИСКОМЫХ ЭЛЕМЕНТОВ НЕСКОЛЬКО

Если в массиве искомых элементов не один, а несколько, какой индекс вернёт метод `indexOf` :

```
const numbers = [ 1, 1, 2, 2, 2, 3, 3 ];
console.log(numbers.indexOf(2)); // 2
```

Так как перебор элементов ведется с первого, и при первом же соответствии поиск прекращается, то мы получим индекс первого элемента соответствующего искомому.

ПОИСК СПРАВА НАЛЕВО

Если в массиве искомых элементов не один, и нам нужен индекс последнего из них, можем воспользоваться методом `lastIndexOf`:

```
const numbers = [ 1, 1, 2, 2, 2, 3, 3 ];
console.log(numbers.lastIndexOf(2)); // 4
```

Метод `lastIndexOf` полностью идентичен методу `indexOf`, но возвращает позицию последнего элемента.

ФУНКЦИИ ВЫСШЕГО ПОРЯДКА

>ФУНКЦИИ ВЫСШЕГО ПОРЯДКА

>Функция, принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата.

High ordered function или *HOF* – функция высшего порядка в переводе с английского.

ПРИНИМАЕМ ФУНКЦИЮ В КАЧЕСТВЕ АРГУМЕНТА

Для начала разберем простой пример:

```
1 function execute(func) {
2     func();
3 }
4
5 function hello() {
6     console.log('Привет, я функция!');
7 }
8
9 execute(hello);
10 execute(function () {
11     console.log('Я функциональное выражение!');
12 });
```

ФУНКЦИИ ПЕРЕДАЮТСЯ БЕЗ СКОБОК

Обратите внимание, что функцию `hello` мы передаем просто по имени.

```
1 function execute(func) {  
2     func();  
3 }  
4  
5 function hello() {  
6     console.log('Привет, я функция!');  
7 }  
8  
9 execute(hello());
```

Мы передали результата выполнения `hello`. Так как функция `hello` ничего не возвращает, то внутри функции `execute` переменная `func` будет указывать на `undefined`, а не на функцию `hello` как ожидалось изначально.

ИМЯ ФУНКЦИИ

Функция тоже объект. И поэтому у неё есть некоторые свойства и методы.

Например свойство `name` – содержит строку, название функции:

```
1 function mult(a, b) {  
2     return a * b;  
3 }  
4  
5 console.log(mult.name); // mult
```

ВОЗВРАЩАЕМ ФУНКЦИЮ ИЗ ФУНКЦИИ (ДЕМО)

Разберём пару функций

```
1 function example1(){
2     let innerExample1Func = () => "innerExample1Func_result";
3     return innerExample1Func();
4 }
5
6 function example2(){
7     let innerExample2Func = () => "innerExample2Func_result";
8     return innerExample2Func;
9 }
```

Какие значения значения будут получены при вызове функций
example1 и example2 ?

ВОЗВРАЩАЕМ ФУНКЦИЮ ИЗ ФУНКЦИИ

```
> function example1(){
    let innerExample1Func = () => "innerExample1Func_result";
    return innerExample1Func();
}

function example2(){
    let innerExample2Func = () => "innerExample2Func_result";
    return innerExample2Func;
}
< undefined

> example1()
< "innerExample1Func_result"
> example2()
< () => "innerExample2Func_result"
```

ВЫЗЫВАЕМ ПОЛУЧЕННУЮ ФУНКЦИЮ

Так как из функции возвращается функция, то вызывать её можно просто дописав `()`.

```
> function example2(){
    return () => "innerExample2Func_result";
}
let resultFunc = example2();
< undefined
> resultFunc();
< "innerExample2Func_result"
> // без переменной
example2()();
< "innerExample2Func_result"
>
```

В первом вызове результат функции `example2` сохраняется в переменную `resultFunc`. Затем `resultFunc` вызывается как функция. Во втором вызове результат функции `example2` не сохраняется, а сразу вызывается как функция.

РАСШИРЕННЫЙ ПОИСК В МАССИВЕ

ПОИСК ИНДЕКСА ЭЛЕМЕНТА В БОЛЕЕ СВОБОДНОЙ ФОРМЕ

Что если мы хотим найти индекс первого четного элемента в массиве?

```
const numbers = [ 127, 41, 454, 296, 489 ];
```

Метод `indexOf` тут явно не поможет, потому что мы не знаем, какое это будет число. Знаем только, что оно будет четное. Попробуем перебрать массив в цикле. Как только мы нашли четное число, сохраняем его в переменную и прерываем цикл.

```
1 let evenIndex = -1;
2 for (let i in numbers) {
3   const number = numbers[i];
4   if (number % 2 === 0) {
5     evenIndex = i;
6     break;
7   }
8 }
9 console.log(evenIndex); // 2
```

ИМПЕРАТИВНОСТЬ И ДЕКЛАРАТИВНОСТЬ

Императивный код – полностью описывает каждое необходимое действие, чтобы достигнуть поставленной задачи.

Декларативный код – описывает задачу и ожидаемый результат. Как должен быть достигнут ожидаемый результат, может быть вообще не описано.

Минус императивного кода – за командами не всегда ясна решаемая задача.
Минус декларативного кода – непонятно, как достигается поставленный результат.

Декларативный код легче читать, поэтому всегда стремитесь не просто решить задачу, но и сделать решение более декларативным.

В JavaScript сделать код более декларативным помогают создаваемые программистом функции и использование стандартных возможностей языка. Часто использование своего решения вместо использования средства языка называют «велосипедом».

МЕТОД `findIndex`

В JavaScript у массива есть метод `findIndex`, который позволяет найти индекс искомого элемента, описав условие функцией.

Метод принимает в качестве аргумента функцию проверки, применяет её к каждому элементу массива до тех пор, пока функция проверки не вернет истину и возвращает текущий индекс элемента.

`findIndex` является функцией высшего порядка. Кроме того, что вместо значения искомого элемента он принимает функцию проверки, в остальном он идентичен методу `indexOf`.

ИЗБАВЛЯЕМСЯ ОТ ВЕЛОСИПЕДОВ

Функция проверки должна принимать элемент массива и возвращать истину или ложь, поэтому наша функция `isEven` отлично подойдет в качестве функции проверки:

```
1 const numbers = [ 127, 41, 454, 296, 489 ];
2 const numbersWithoutEven = [ 127, 41, 365, 7, 489 ];
3
4 function isEven(number) {
5   return number % 2 === 0;
6 }
7
8 console.log(numbers.findIndex(isEven)); // 2
9 console.log(numbersWithoutEven .findIndex(isEven)); // -1
```

Обратите внимание: несмотря на то, что четных элементов в массиве несколько, возвращается индекс первого.

Метода `findLastIndex` у массивов нет. Потому что всегда можно воспользоваться методом `reverse`, который переставляет элементы массива в обратном порядке.

А ЧТО ЕСЛИ ХОТИМ НАЙТИ САМО ЧЕТНОЕ ЧИСЛО

Воспользуемся методом `find`, который очень похож на `findIndex`, только возвращает не индекс, а сам элемент:

```
1 const numbers = [ 127, 41, 454, 296, 489 ];
2 const numbersWithoutEven = [ 127, 41, 365, 7, 489 ];
3
4
5 function isEven(number) {
6   return number % 2 === 0;
7 }
8
9 console.log(numbers.find(isEven)); // 454
10 console.log(numbersWithoutEven.find(isEven)); // undefined
```

ЧАСТАЯ ОШИБКА С `find`

```
1 const numbers = [ 127, 0, 41, 365, 7, 489 ];  
2  
3 function isEven(number) {  
4     return number % 2 === 0;  
5 }  
6  
7 if (!numbers.find(isEven)) {  
8     // Метод `find` на самом деле нашел четное число 0, но !0 === true  
9     console.log('В массиве все числа нечетные');  
10 }  
11  
12 if (numbers.find(isEven) === undefined) {  
13     // Правильная проверка  
14     console.log('В массиве все числа нечетные');
```

«ХОТЯ БЫ ОДИН»

У массива есть специальный метод `some` (*хотя бы один* в переводе с английского). Он очень похож на `find`, но возвращает не сам элемент, а истину, если есть хотя бы один элемент удовлетворяющий условию. Чем также похож на `includes`.

Используем его для нашего примера:

```
1 const numbers = [ 127, 0, 41, 365, 7, 489 ];  
2  
3 function isEven(number) {  
4     return number % 2 === 0;  
5 }  
6  
7 console.log(numbers.some(isEven)); // true
```

«КАЖДЫЙ»

Используем метод `every` (каждый в переводе с английского), который похож на `some`, но возвращает истину только если все элементы массива удовлетворяют требованиям:

```
1 function isEven(number) {  
2     return number % 2 === 0;  
3 }  
4  
5 function isEvenArray(arr) {  
6     return arr.every(isEven);  
7 }  
8  
9 console.log(isEvenArray([ 24, 42, 176 ])); // true  
10 console.log(isEvenArray([ 24, 41, 176 ])); // false
```

«ХОТЯ БЫ ОДИН» И «КАЖДЫЙ»

Очень часто поиск элементов в массиве сводится к двум простым проверкам:

- есть ли в массиве хотя бы один элемент, соответствующий требованиям;
- соответствуют ли все элементы массива требованиям.

Для решения подобных задач можно использовать `find` или `findIndex`. Но гораздо удобнее использовать специальные методы `some` и `every`. Их работа похожа на `find`, но они сразу возвращают истину или лож, что проще использовать в `if`.

ПОИСК В МАССИВЕ ОБЪЕКТОВ

Допустим, у нас есть массив сотрудников:

```
1 | const employees = [
2 |   { name: 'Мария', department: 'IT', salary: 75000 },
3 |   { name: 'Иван', department: 'Продажи', salary: 55000 },
4 |   { name: 'Николай', department: 'IT', salary: 92000 },
5 |   { name: 'Мария', department: 'Маркетинг', salary: 35000 }
6 | ];
```

Давайте решим следующие простые задачи:

1. Найдем Марию из отдела IT.
2. Выясним, есть ли у кого-то зарплата более 90000.
3. Проверим, все ли получают более 50000.

РЕШЕНИЕ

Работа с массивом объектов ничем не отличается от работы с массивом чисел, просто в функции проверки мы будем работать с объектом, а не числом. Для решения задач нам потребуются:

1. `find`
2. `some`
3. `every`

РЕШЕНИЕ

```
1 const task1 = employees.find(employee =>
2     employee.name === 'Мария' && employee.department === 'IT';
3 );
4
5 const task2 = employees.some(employee =>
6     employee.salary > 90000;
7 );
8
9 const task3 = employees.every(employee =>
10    employee.salary > 50000;
11 );
```

НАЙТИ ВСЕ ЭЛЕМЕНТЫ, УДОВЛЕТВОРЯЮЩИЕ УСЛОВИЮ

Что если нам потребуется найти всех сотрудников по имени **Мария** ?

МЕТОД `filter`

Метод `filter` очень похож на `find`, только в отличии от него всегда возвращает новый массив, в который помещает все элементы исходного массива, удовлетворяющие требованиям заданным в функции проверки.

Грубо говоря, каждый элемент, для которого функция проверки вернет истину, будет добавлен в массив. Если таких элементов нет, то массив будет пустой.

НАХОДИМ ВСЕХ МАРИЙ

```
1 const maries = employees.filter(employee => employee.name === 'Мария');
2 console.log(maries);
3 // [
4 //   { name: 'Мария', department: 'IT', salary: 75000 },
5 //   { name: 'Мария', department: 'Маркетинг', salary: 35000 },
6 // ]
```

ЗАРПЛАТА МЕНЬШЕ 90000 НЕ В IT

```
1 const noItBelow90K = employees.filter(employee => employee.department !== 'IT' && employee.salary < 90000);
2 console.log(noItBelow90K);
3 // [
4 //   { name: 'Иван', department: 'Продажи', salary: 55000 },
5 //   { name: 'Мария', department: 'Маркетинг', salary: 35000 },
6 // ]
```

ВСЕГДА МАССИВ

Даже если ничего не найдено или найден один элемент:

```
1 const over90K = employees
2   .filter(employee => employee.salary > 90000);
3
4 console.log(over90K);
5 // [ { name: 'Николай', department: 'IT', salary: 92000 } ]
6
7 const noItOver90K = employees
8   .filter(employee => employee.department !== 'IT' && employee.salary > 90000);
9
10 console.log(noItOver90K); // []
```

ПРОВЕРКА НА ПОНИМАНИЕ filter

Вопрос: Что будет выведено на консоль?

Пример 1:

```
console.log([1, 2, 3, 4].filter(item => true));
```

Пример 2:

```
console.log([1, 2, 3, 4].filter(item => item - 2));
```

ПРОВЕРКА НА ПОНИМАНИЕ `filter`

Пример 1.

Так как функция проверки всегда возвращает истину, получим полную копию массива:

```
// [ 1, 2, 3, 4 ]
```

Пример 2.

Так как `filter` ждет от функции истину или ложь, а функция проверки возвращает числа, то числа приводятся к булевому значению. Для элемента `2` функция проверки вернет `0`, что будет приведено ко лжи, остальные — к истине:

```
[ 1, 3, 4 ]
```

ПРЕОБРАЗОВАНИЕ МАССИВА

СОЗДАНИЕ НОВОГО МАССИВА

Допустим, у нас есть массив, который содержит интервалы времени в секундах:

```
const timeIntervals = [1800, 3600, 86400];
```

Нам для дальнейших расчетов нужно перевести эти интервалы из секунд в часы.

ИСПОЛЬЗУЕМ ЦИКЛ

Переберем все интервалы исходного массива, переведем каждый в часы и поместим в новый массив:

```
1 const timeIntervals = [1800, 3600, 86400];
2 const secondsInHour = 3600;
3
4 function secondsToHours(sec) {
5   return sec / secondsInHour;
6 }
7
8 const timeIntervalsInHours = [];
9 for (const interval of timeIntervals) {
10   const hours = secondsToHours(interval);
11   timeIntervalsInHours.push(hours);
12 }
13 console.log(timeIntervalsInHours); // [ 0.5, 1, 24 ]
```

В JavaScript есть возможность сделать еще лучше.

МЕТОД МАССИВА `map`

Создает новый массив, поместив в него результат вызова переданной функции для каждого элемента исходного массива.

Метод `map` – это функция высшего порядка, так как принимает функцию первым аргументом. Именно эта переданная в `map` функция будет вызвана для каждого элемента массива. Назовем её преобразователем.

ПРЕОБРАЗУЕМ ИНТЕРВАЛЫ С ПОМОЩЬЮ map

Метод `map` принимает функцию, создает и возвращает новый массив:

```
1 const timeIntervals = [1800, 3600, 86400];
2 const secondsInHour = 3600;
3
4 const timeIntervalsInHours = timeIntervals.map(sec => sec / secondsInHour);
5
6 console.log(timeIntervalsInHours); // [ 0.5, 1, 24 ]
```

Если прочитать код решения и перевести на русский, то будет похоже на

преобразовать `timeIntervals` с помощью функции
преобразователя и поместить результат в `timeIntervalsInHours`.

ОСОБЕННОСТИ ИСПОЛЬЗОВАНИЯ `map`

При использовании метода массива `map` нужно обязательно учитывать ряд важных особенностей:

1. размер преобразованного массива всегда равен размеру исходного массива;
2. игнорирование результата метода `map` хоть и не является ошибкой, но обычно указывает на неправильное использование этого метода.

Разберём на примерах.

РАЗМЕР МАССИВА РАВЕН ИСХОДНОМУ

Попробуем возвести в квадрат только четные числа, а нечетные проигнорировать:

```
1 const numbers = [1, 2, 3, 4, 5];
2
3 const sqEven = numbers.map(function (number) {
4     if (number % 2 === 0) {
5         return number * number;
6     }
7 });
8
9 console.log(sqEven); // [ undefined, 4, undefined, 16, undefined ]
```

В JavaScript функция всегда возвращает результат, даже если во второй ветке нет `return`. Поэтому вместо нечетных чисел у нас `undefined`.

Используйте `map` совместно с `filter` для решения этой задачи.

КВАДРАТЫ ЧЕТНЫХ ЧИСЕЛ (ДЕМО)

Просто откинем `undefined` после, собрав вызовы `map` и `filter` в цепочку:

```
1 const numbers = [1, 2, 3, 4, 5];
2
3 const sqEven = numbers
4   .map(function (number) {
5     if (number % 2 === 0) {
6       return number * number;
7     }
8   })
9   .filter(item => item !== undefined);
10
11 console.log(sqEven); // [ 4, 16 ]
```

ПОРЯДОК ИМЕЕТ ЗНАЧЕНИЕ

А что если сначала отбросить нечетные числа а потом уже возвести полученный массив в квадрат:

```
1 const numbers = [1, 2, 3, 4, 5];  
2  
3 const sqEven = numbers  
4   .filter(number => number % 2 === 0)  
5   .map(number => number * number);  
6  
7 console.log(sqEven); // [ 4, 16 ]
```

Результат тот же, но посмотрите как решение стало выглядеть проще, логичнее и декларативнее. Его можно читать как простой текст.

ИГНОРИРОВАНИЕ РЕЗУЛЬТАТА

Типичный пример неправильного использования `map`:

```
[1, 2, 3, 4, 5].map(item => console.log(item));
```

Видно, что массив преобразуется с помощью функции-стрелки, но результат нам не важен. Потому что в данном случае мы используем `map` для простого перебора. Правильнее это переписать с помощью метода `forEach` или цикла `for-of`:

```
[1, 2, 3, 4, 5].forEach(item => console.log(item));
```

ЧЕМУ МЫ НАУЧИЛИСЬ?

1. Изучили основные методы массивов;
2. Изучили функции, которые могут быть аргументами функций;
3. Сокращенную запись функций (функции-стрелки);
4. Изучили методы преобразования массивов.

ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаем в чате Slack!
- Задачи можно сдавать по частям.
- Зачет по домашней работе проставляется после того, как приняты **все задачи**.



Спасибо за внимание! Время задавать вопросы

ЕВГЕНИЙ ШЕК



shek.e.q@yandex.ru



[@usertel](https://t.me/usertel)