нетология

Работа с массивами

Антон
Степанов

Антон Степанов

Ведущий фронтэнд разработчик в StepIntegrator

@anton_mesmer

План занятия

1. Итерация по массиву
2. Поиск в массиве
3. Расширенный поиск в массиве
4. Преобразование массива

Вспомним то, что мы уже знаем

Какие методы вы помните?

Вы уже знакомы с **массивами** и некоторыми методами. Проверьте себя:

```
const arr = ["Ann", "Helen", "Fox"];
arr.push("Richard");
arr.push(123);
arr.push("Jane");
arr.pop();
arr.shift();
arr.unshift("Olga");
console.log(arr);
let arr2 = arr.slice(1, 3);
console.log(arr2);

arr.splice(1, 2, "some val");
console.log(arr);

let arr3 = arr.concat(["learn", "js"]);
console.log(arr3);
```

Каков будет результат?

Какие методы вы помните?

Вы уже знакомы с **массивами** и некоторыми методами. Проверьте себя:

```
const arr = ["Ann", "Helen", "Fox"];
arr.push("Richard");
arr.push(123);
arr.push("Jane");
arr.pop();
arr.shift();
arr.unshift("Olga");
console.log(arr);
let arr2 = arr.slice(1, 3);
console.log(arr2);

arr.splice(1, 2, "some val");
console.log(arr);

let arr3 = arr.concat(["learn", "js"]);
console.log(arr3);
```

Ответы

```
["Olga", "Helen", "Fox", "Richard",
123][("Helen", "Fox")][
  ("Olga", "some val", "Richard", 123)
][("Olga", "some val", "Richard", 123,
"learn", "js")]
```

Вспомним константы

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

```
const numbers = [1, 2, 3];
numbers.shift();
numbers.pop();
console.log(numbers);
numbers = [3, 2, 6]; // Ошибка
```

Как видим, *shift* и *pop* отработали и внесли изменения в массив.

Изменять массивы можно. **Запрещено только присваивание**.

Мы выяснили, что массив, сохранённый в константе, можно изменять с помощью его **методов**

Мутабельность и иммутабельность

Объекты, значение которых может измениться, или свойства, которые можно менять напрямую, являются изменяемыми — то есть **мутабельными** *(от слов **мутация**, **мутант**)*.

Запомните ещё раз: оператор *const* не делает объекты и массивы **иммутабельными** *(для этого есть метод Object.freeze(), const 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 неё передать не массив?

```
function logArray(arr) {
  console.log(`Массив размером ${arr.length}`)
  for (let item of arr) {
    console.log(`${item} (${typeof item})`)
  }
}

logArray(12);
```

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 неё передать не массив?

```
function logArray(arr) {
  console.log(`Массив размером ${arr.length}`)
  for (let item of arr) {
    console.log(`${item} (${typeof item})`)
  }
}

logArray(12);
```

```
// Массив размером undefined
// TypeError: arr is not iterable
```

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

Проверка на массив: Array.isArray

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

```
function checkArr(arr) {
  if (!Array.isArray(arr)) {
    console.log(`Массив размером ${arr.length}`)
  } else {
    console.log("Это не массив");
  }
}
```

Проверка на массив: Array.isArray

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

```
function checkArr(arr) {
  if (!Array.isArray(arr)) {
    console.log(`Массив размером ${arr.length}`)
  } else {
    console.log("Это не массив");
  }
}
```

Такого же результата можно добиться, используя:

```
if (arr instanceof Array) {
  console.log("Это массив");
}
```

Преобразование в массив с помощью Array.from

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

```
const str = "Не массив!";

const simbols = Array.from(str);

console.log(simbols);
```

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**:

```
function logArgs() {}

logArgs(1, 2, 3);
```

Как в теле функции **обратиться к аргументам**, переданным при вызове функции, если их количество заранее неизвестно?

Массивоподобный объект *arguments*

Внутри каждой функции доступна **переменная** *arguments*, которая предоставляет альтернативный доступ ко всем переданным в функцию аргументам (альтернатива аргументу *rest*):

```
function logArgs() {
  console.log(arguments); // Arguments[1,2,3]
  console.log(arguments.length); // 3
  console.log(Array.isArray(arguments)); // false
  console.log(arguments.join(", ")); // Uncaught TypeError: arguments.join is not a function
}

logArgs(1, 2, 3);
```

Мы видим, что *arguments* **не является** массивом, хотя и очень на него похожа.

И у неё нет метода *join*

Преобразуем *arguments* в массив

Вот здесь мы и применим **функцию** *Array.from*, чтобы получить полноценный массив:

```
function logArgs() {
  console.log(Array.from(arguments).join(", "));
}

logArgs(1, 2, 3); // 1, 2, 3
```

Преобразование **объекта** *arguments* в **массив** имеет смысл, только если нам нужны возможности массива, которых в объекте нет

Итерация по массиву

Итерация по массиву

Любая задача работы с массивом — поиск, преобразования, фильтрация — может быть решена **итерацией по массиву**. В этом смысле этот способ универсальный

Способы итерации

1. Уже знакомый нам универсальный цикл *for*:

```
const arr = [30, 10, 50, 1, 31, 178, 15];

for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}
```

Способы итерации

1. Уже знакомый нам универсальный цикл *for*
2. Метод *forEach*:

```
arr.forEach((item, idx, arr) => console.log(item));
```

Способы итерации

1. Уже знакомый нам универсальный цикл *for*
2. Метод *forEach*
3. Тот же результат с помощью *for...of*:

```
for (const item of arr) {
  console.log(item);
}
```

Способы итерации

1. Уже знакомый нам универсальный цикл *for*
2. Метод *forEach*
3. Тот же результат с помощью *for...of*
4. Тот же результат с помощью *for...in*:

```
for (const i in arr) {
  console.log(arr[i]);
}
```

Способы итерации

1. Уже знакомый нам универсальный цикл *for*
2. Метод *forEach*
3. Тот же результат с помощью *for...of*
4. Тот же результат с помощью *for...in*

Метод *forEach* самый быстрый. Также можно воспользоваться циклом *while*

Сравнение скорости можно посмотреть здесь: https://jsbench.me/yhkngi7lx9/1

Поиск в массиве

Индекс искомого элемента

Допустим, у нас есть **массив имён**, 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*. Он строго сравнивает аргумент с каждым элементом массива и возвращает индекс первого элемента, сравнение с которым вернёт истину:

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

console.log(names.indexOf("Иван")); // 3
console.log(names.indexOf("Егор")); // 1
console.log(names.indexOf("Маша")); // -1
```

Обратите внимание, что метод возвращает **индекс**, а не **номер**.

Индексы начинаются с *0*. Метод *indexOf* вернёт *-1*, если искомого элемента нет

Частая ошибка с *indexOf*

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

```
function pushOnce(arr, item) {
  if (!arr.indexOf(item)) {
    arr.push(item);
  }
}

const uniqNumbers = [];
pushOnce(uniqNumbers, 1);
pushOnce(uniqNumbers, 2);
pushOnce(uniqNumbers, 2);
pushOnce(uniqNumbers, 1);

console.log(uniqNumbers); // []
```

Массив пуст, хотя там должно быть два элемента.

Почему?

Правильная проверка наличия

Когда элемент не найден, *indexOf* возвращает *-1* — именно это и нужно проверять в функции *pushOnce*:

```
function pushOnce(arr, item) {
  if (arr.indexOf(item) === -1) {
    arr.push(item);
  }
}

const uniqNumbers = [];
pushOnce(uniqNumbers, 1);
pushOnce(uniqNumbers, 2);
pushOnce(uniqNumbers, 2);
pushOnce(uniqNumbers, 1);

console.log(uniqNumbers); // [1,2]
```

Теперь функция работает, как задумывалось

Лучшая проверка наличия

Метод *indexOf* часто используется в условиях, и многие 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, а если его нет — возвращает **false**

Лучшая проверка наличия

Перепишем функцию *pushOnce*, используя более подходящий метод:

```
function pushOnce(arr, item) {
  if (!arr.includes(item)) {
    arr.push(item);
  }
}

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

Если искомых элементов несколько

Если в массиве искомых элементов несколько, какой индекс вернет 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 искомый элемент 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 *indexOf*, но возвращает позицию последнего элемента

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* здесь явно не поможет, потому что мы не знаем, какое это будет число. Знаем только, что оно будет чётное

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];
```

Попробуем перебрать массив **в цикле**. Как только мы нашли чётное число, сохраняем его в переменную и прерываем цикл

```
let evenIndex = -1
for (let i in numbers) {
  const number = numbers[i];
  if (number % 2 === 0) {
    evenIndex = i;
    break
  }
}

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 результат, может быть вообще не описано

Императивность и декларативность

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

- Минус императивного кода — за командами не всегда ясна решаемая задача

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

- Минус декларативного кода — непонятно, как достигается поставленный результат

Императивность и декларативность

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 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 и **применяет** её к каждому элементу массива до тех пор, пока функция проверки не вернёт истину, затем **возвращает** текущий индекс элемента

Метод *findIndex*

FindIndex — **функция высшего порядка**. Кроме того, что вместо значения искомого элемента метод принимает функцию проверки, в остальном он идентичен методу *indexOf*.

Ищем первый элемент со значением 18 и более:

```
var ages = [3, 10, 18, 20];
function checkAdult(age) {
  return age >= 18;
}
const adult = ages.findIndex(checkAdult);
```

Запись короче с использованием стрелочной функции:

```
const index = findIndex((item,idx) => age > 18);
```

Избавляемся от «велосипедов»

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

```
const numbers = [127, 41, 454, 296, 489];
const numbersWithoutEven = [127, 41, 365, 7, 489];

function isEven(number) {
  return number % 2 === 0;
}

console.log(numbers.findIndex(isEven)); // 2
console.log(numbersWithoutEven.findIndex(isEven)); // -1
```

Обратите внимание: несмотря на то, что чётных 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 чётное число?

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

```
const numbers = [127, 41, 454, 296, 489];
const numbersWithoutEven = [127, 41, 365, 7, 489];

function isEven(number) {
  return number % 2 === 0;
}

console.log(numbers.findIndex(isEven)); // 2
console.log(numbersWithoutEven.findIndex(isEven)); // -1
```

Частая ошибка с *find*

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

```
const numbers = [127, 0, 41, 365, 7, 489];

function isEven(number) {
  return number % 2 === 0;
}

if (!numbers.find(isEven)) {
  // Метод `find` на самом деле нашел чётное число 0, но !0 === true
  console.log("В массиве все числа нечетные");
}

if (numbers.find(isEven) === undefined) {
  // Правильная проверка
  console.log("В массиве все числа нечетные");
}
```

«Хотя бы один» — *some*

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 условие. Этим он также похож на *includes*.

Используем его для нашего примера (воспользуемся стрелочной функцией):

```
const numbers = [127, 0, 41, 365, 7, 489];

console.log(numbers.some((item)=> item % 2 === 0); // true
```

«Каждый» — *every*

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

```
function isEven(number) {
  return number % 2 === 0;
}

function isEvenArray(arr) {
  return arr.every(isEven);
}

console.log(isEvenArray([24, 42, 176])); // true
console.log(isEvenArray([24, 41, 176])); // false
```

«Хотя бы один» и «каждый»

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

1. есть ли в массиве **хотя бы один элемент**, соответствующий требованиям;
2. 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 ложь, что проще использовать в *if*.

Поиск в массиве объектов

Допустим, у нас есть **массив** сотрудников:

```
const employees = [
  { name: "Мария", department: "IT", salary: 75000 },
  { name: "Иван", department: "Продажи", salary: 55000 },
  { name: "Николай", department: "IT", salary: 92000 },
  { name: "Мария", department: "Маркетинг", salary: 35000 },
]
```

Давайте решим следующие простые **задачи**:

1. **найдём** Марию из отдела IT;
2. **выясним**, есть ли у кого-то зарплата более 90 тыс;
3. **проверим**, все ли получают более 50 тыс.

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 *find*, *some* и *every*:

```
const task1 = employees.find(employee =>
  employee.name === 'Мария' && employee.department === 'IT';
);

const task2 = employees.some(employee =>
  employee.salary > 90000;
);

const task3 = employees.every(employee =>
  employee.salary > 50000;
);
```

Найти все элементы, удовлетворяющие условие

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

```
const employees = [
  { name: "Мария", department: "IT", salary: 75000 },
  { name: "Иван", department: "Продажи", salary: 55000 },
  { name: "Николай", department: "IT", salary: 92000 },
  { name: "Мария", department: "Маркетинг", salary: 35000 },
]
```

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 требования, заданные в функции проверки.

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

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

Метод *filter*

Находим всех *Марий*:

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

Метод *filter*

Зарплата меньше 90 тыс. не в IT:

```
const noItBelow90K = employees.filter(
  (employee) => employee.department !== "IT" && employee.salary < 90000
);

console.log(noItBelow90K);
// [
//   { name: 'Иван', department: 'Продажи', salary: 55000 },
//   { name: 'Мария', department: 'Маркетинг', salary: 35000 },
// ]
```

Filter всегда возвращает массив

Даже если **ничего не найдено** или **найден один** элемент:

```
const over90K = employees.filter((employee) => employee.salary > 90000);

console.log(over90K);
// [ { name: 'Николай', department: 'IT', salary: 92000 } ]

const noItOver90K = employees.filter(
  (employee) => employee.department !== "IT" && employee.salary > 90000
);

console.log(noItOver90K); // []
```

Проверка на понимание filter

Вопрос: что будет выведено в 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

Вопрос: что будет выведено в консоль?

пример 1

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

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

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

Проверка на понимание filter

Вопрос: что будет выведено в консоль?

пример 2

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

Ответ: так как *filter* ждёт от функции истину или ложь, а функция проверки возвращает числа, то числа приводятся к **нулевому значению**.

Для элемента 2 функция проверки вернет 0, то есть ложь, для остальных - истину:

```
// [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 расчётов нужно перевести эти интервалы из секунд в часы.

Используем цикл

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

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

function secondsToHours(sec) {
  return sec / secondsInHour;
}

const timeIntervalsInHours = [];
for (const interval of timeIntervals) {
  const hours = secondsToHours(interval);
  timeIntervalsInHours.push(hours);
}
console.log(timeIntervalsInHours); // [ 0.5, 1, 24 ]
```

В JavaScript есть возможность сделать ещё 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, создаёт и возвращает **новый массив**:

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

const timeIntervalsInHours = timeIntervals.map((sec) => sec / secondsInHour);

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**:

- **размер** преобразованного массива всегда равен размеру исходного массива
- **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 чётные числа, а нечётные проигнорировать:

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

const sqEven = numbers.map(function (number) {
  if (number % 2 === 0) {
    return number * number;
  }
})

console.log(sqEven); // [ undefined, 4, undefined, 16, undefined ]
```

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

Квадраты чётных чисел (демо)

Используйте *map* совместно с *filter* для решения этой задачи. Просто откинем *undefined* после, собрав вызовы *map* и *filter* в цепочку:

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

const sqEven = numbers
  .map(function (number) {
    if (number % 2 === 0) {
      return number * number;
    }
  })
  .filter(item => item !== undefined);

console.log(sqEven);// [ 4, 16 ]
```

Порядок имеет значение

А что, если сначала отбросить нечётные числа, а потом уже возвести полученный массив в квадрат?

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

const sqEven = numbers
  .filter((number) => number % 2 === 0)
  .map((number) => number * number);

console.log(sqEven); // [ 4, 16 ]
```

Результат тот же, но посмотрите, насколько 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));
```

Клонирование массива

Копирование *(клонирование)* **массива** — весьма частая задача, которую можно решить несколькими способами:

```
const arr = ["Hello", "new", "world!"];
const clone1 = arr.slice();
const clone2 = [...arr];

const clone3 = [];
for (let i = 0; i < arr.length; i++) clone3.push(arr[i]);

let clone4 = arr.map((item) => item); //Как работает

//Если массив содержит вложенные массивы|объекты, наиболее удобный способ
let clone5 = JSON.parse(JSON.stringify(arr));
```

Сортировка массива

Метод *sort* массива является **мутабельным** и изменяет исходный массив:

```
const arr = ["Boris", "Cesar", "Ann"];
arr.sort() // arr === ["Ann","Boris", "Cesar"];
```

Метод *sort* допускает аргумент — **функцию сортировки**:

```
const myarr = [25, 8, 7, 41];
myarr.sort((a,b) => (a - b));
// Передаем внутрь стрелочную функцию.
// А как бы это выглядело с обычной функцией?
// Как отсортировать в обратном порядке?
// Как отсортировать немутабельно?
```

Ответ

```
const myarr= [25, 8, 7, 41];
const newArr = myarr.slice;
myarr.sort(function(a,b) { return (b - a) });
// Иммутабельно. В обратном порядке.
// Обычная функция в качестве функции сортировки
```

Универсальный метод reduce

Метод *reduce()* применяет функцию *reducer* **к каждому элементу** массива *(слева направо)*, возвращая одно результирующее значение, которое в свою очередь может быть массивом. В этом смысле он **универсален**:

```
// arr.reduce(reducer, initialAcc);
const arr = [10, 20, 15, 35, 5, 45, 60];
arr.reduce((acc, item, idx, arr) => {
  // Будет ровно arr.length шагов. На каждом шаге;
  // item - элемент;
  // idx - индекс элемента массива;
  // arr - исходный массив;
  // acc - аккумулятор, он же результат.

  return // в acc для следующего шага будет записано то, что в return;
}, initialAcc);
//Если initialAcc не передан, то берется первый элемент массива
```

Универсальный метод reduce: пример

Посчитаем количество четных элементов массива:

```
const arr = [10, 20, 15, 35, 5, 45, 60];

const result = arr.reduce((acc, item, idx, arr) => {
  let newAcc = acc;
  if (item % 2 === 0) {
    newAcc++;
  }

  return newAcc;
}, 0) // начальное значение счетчика 0
```

Универсальный метод reduce: пример

Посчитаем среднее арифметическое:

```
const numbers = [10, 20, 15, 35, 5, 45, 60];

const average = numbers.reduce((acc, item, idx, arr) => {
  acc += item;
  if (idx === arr.length - 1) {
    return acc / arr.length;
  } else {
    return acc;
  }
})

console.log(average);
```

Универсальный метод reduce: пример

Часть методов для работы с массивами **аналогично работают со строками**:

- *concat*
- *includes*
- *indexOf*
- *slice*

Итоги

Чему мы научились?

- Константы и массивы, мутабельность и иммутабельность
- Поиск в массиве
- Функции высшего порядка
- Расширенный поиск в массиве
- Методы массива

Домашнее задание

Давайте посмотрим ваше домашнее задание.

- **Вопросы** по домашней работе задаем в группе в Slack
- Задачи можно сдавать **по частям**
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**

нетология

Задавайте вопросы и пишите отзыв о лекции!

Антон Степанов

@anton_mesmer