

Тема 9. Массивы.



Хекслет Колледж

Цель занятия:

Сформировать у студентов понимание массивов как структуры данных, научить создавать массивы, получать доступ к элементам и выполнять базовые операции обработки данных.

Учебные вопросы:

1. Понятие массива в JavaScript
2. Создание массивов
3. Доступ к элементам массива
4. Изменение элементов массива
5. Основные методы массивов
6. Перебор массивов
7. Методы перебора массивов

1. Понятие массива в JavaScript

Массив — это структура данных, которая позволяет хранить несколько значений в одной переменной.

Массив упрощает хранение, обработку и перебор данных.

Назначение массивов

Массивы используются, когда нужно работать с набором однотипных или связанных данных, например:

- список чисел;
- список имён пользователей;
- список товаров;
- набор оценок;
- массив строк, логических значений и т.д.

Особенности массивов в JavaScript

Массив может содержать элементы разных типов:

- **let data = [10, "текст", true];**

Каждый элемент массива имеет порядковый номер — индекс.

Индексация начинается с нуля.

Массив является объектом специального типа.

Структура массива

```
let fruits = ["apple", "banana", "orange"];
```

- **fruits** — имя массива
- "apple", "banana", "orange" — элементы массива
- 0, 1, 2 — индексы элементов

Зачем использовать массивы

Массивы позволяют:

- хранить данные компактно;
- работать с большими наборами данных;
- выполнять операции сортировки, фильтрации и преобразования;
- удобно обрабатывать данные в циклах.

Вывод:

- Массив — это упорядоченная коллекция значений.
- Используется для хранения нескольких значений в одной переменной.
- Элементы массива имеют индексы, начинающиеся с нуля.
- Массивы — одна из ключевых структур данных в JavaScript.

2. Создание массивов

В JavaScript существует несколько способов создать массив, но на практике используется преимущественно один основной.

1. Литерал массива [] (основной способ)

Самый простой и рекомендуемый способ создания массива — с помощью квадратных скобок:

```
let numbers = [1, 2, 3, 4];  
let names = ["Анна", "Иван", "Олег"];
```

Преимущества:

- наглядный и простой синтаксис;
- читаемость кода;
- является стандартом в современной разработке.

Создание пустого массива

```
let items = [];
```

Часто используется, если массив будет заполняться позже, например в цикле.

2. Использование конструктора **Array()**

```
let arr1 = new Array(1, 2, 3);  
let arr2 = Array("a", "b", "c");
```

Особенность:

```
let arr = new Array(5);
```

— создаёт массив длиной 5 без элементов, что часто приводит к путанице.

Поэтому для начинающих не рекомендуется использовать `Array()`.

Вложенные массивы

Массив может содержать другие массивы:

```
let matrix = [  
  [1, 2],  
  [3, 4]  
];
```

Такие структуры используются позже для таблиц, сеток и матриц.

Вывод:

- Массив создаётся с помощью `[]`.
- Можно создать пустой массив или массив с начальными значениями.
- Конструктор `Array()` существует, но для начинающих нежелателен.
- Массивы могут содержать любые типы данных, включая другие массивы.

3. Доступ к элементам массива

Каждый элемент массива имеет индекс — его порядковый номер.

В JavaScript индексация начинается с нуля.

```
let fruits = ["apple", "banana", "orange"];
```

Индекс	Элемент
0	"apple"
1	"banana"
2	"orange"

Получение элемента по индексу

Для доступа к элементу используется запись:

array[index]

Примеры:

console.log(fruits[0]); // apple

console.log(fruits[1]); // banana

Изменение элемента массива

Значение элемента можно изменить по индексу:

```
fruits[1] = "pear";  
console.log(fruits); // ["apple", "pear", "orange"]
```

Получение длины массива

Свойство **length** возвращает количество элементов в массиве:

```
console.log(fruits.length); // 3
```

Важно:

length — не индекс последнего элемента.
Последний индекс всегда равен **length - 1**.

Доступ к последнему элементу массива:

```
let last = fruits[fruits.length - 1];  
console.log(last); // orange
```

Что происходит при обращении к несуществующему индексу?

```
console.log(fruits[10]); // undefined
```

- Ошибки не будет
- Возвращается значение **undefined**

Вывод:

- Элементы массива доступны по индексам.
- Индексация начинается с 0.
- `length` показывает количество элементов.
- Последний элемент имеет индекс `length - 1`.
- Обращение за пределы массива возвращает `undefined`.

4. Изменение элементов массива

Изменение существующих элементов.

Элемент массива можно изменить, обратившись к нему по индексу и присвоив новое значение:

```
let numbers = [10, 20, 30];
```

```
numbers[1] = 25;
```

```
console.log(numbers); // [10, 25, 30]
```

Массив при этом изменяется.

Размер массива остаётся прежним.

Добавление элемента по индексу

Можно добавить новый элемент, указав индекс, которого раньше не было:

```
numbers[3] = 40;  
console.log(numbers); // [10, 25, 30, 40]
```

Если пропустить индекс:

```
numbers[6] = 60;  
console.log(numbers);  
// [10, 25, 30, 40, <2 пустых элемента>, 60]
```

Так делать не рекомендуется, так как появляются «пустые» элементы.

Добавление элемента в конец массива

На практике для добавления элементов используются методы массива (рассматриваются подробно в следующем пункте).

```
numbers[numbers.length] = 50;  
console.log(numbers); // [10, 25, 30, 40, 50]
```

Этот способ возможен, но менее удобен, чем специальные методы.

Удаление элементов

Если присвоить элементу **undefined**, элемент логически удалится, но ячейка останется:

```
numbers[1] = undefined;  
console.log(numbers);
```

Полноценное удаление элементов выполняется с помощью методов массива, которые будут рассмотрены далее.

Вывод:

- Элементы массива можно изменять по индексу.
- Можно добавлять новые элементы, указывая новые индексы.
- Добавление с пропуском индексов приводит к пустым элементам.
- Для добавления и удаления элементов предпочтительно использовать методы массива.
- Массивы в JavaScript являются изменяемыми.

5. Основные методы массивов

Методы массивов позволяют удобно добавлять, удалять и изменять элементы, не работая напрямую с индексами.

push() — добавление в конец массива:

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

```
numbers.push(4);
```

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

Что делает:

- Добавляет элемент(ы) в конец массива.

Аргументы:

- любое количество значений

Что возвращает:

- новую длину массива

pop() — удаление последнего элемента:

```
let last = numbers.pop();
```

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

Что делает:

- Удаляет последний элемент массива.

Аргументы:

- не принимает аргументы

Что возвращает:

- удалённый элемент

unshift() — добавление в начало массива:

```
numbers.unshift(0);  
console.log(numbers); // [0, 1, 2, 3]
```

Что делает:

- Добавляет элементы в начало массива.

Аргументы:

- любое количество значений

Что возвращает:

- новую длину массива

shift() — удаление первого элемента:

```
let first = numbers.shift();
```

```
console.log(numbers); // [1, 2, 3]  
console.log(first);  // 0
```

Что делает:

- Удаляет первый элемент массива.

Аргументы:

- не принимает аргументы

Что возвращает:

- удалённый элемент

slice() — копирование части массива:

```
let arr = [10, 20, 30, 40, 50];  
let part = arr.slice(1, 4);
```

```
console.log(part); // [20, 30, 40]
```

Что делает:

- Создаёт новый массив, копируя элементы исходного.

Аргументы:

- начальный индекс (включительно)
- конечный индекс (не включается)

Что возвращает:

- новый массив

Исходный массив:

- `arr` // не изменяется

splice() — изменение массива

splice() — самый универсальный метод, но и самый сложный.

Общий вид:

array.splice(startIndex, deleteCount, item1, item2, ...)

Аргументы:

- `startIndex` — индекс, с которого начинать
- `deleteCount` — сколько элементов удалить
- `item1, item2, ...` — элементы, которые нужно вставить (необязательно)

Что возвращает:

- массив удалённых элементов
- изменяет исходный массив

Пример 1. Удаление элементов

```
let arr = [1, 2, 3, 4];
```

```
let removed = arr.splice(1, 2);
```

Расшифровка:

- начать с индекса 1
- удалить 2 элемента
- Что удалили: [2, 3]

Результат:

- **arr // [1, 4]**
- **removed // [2, 3]**

Пример 2. Добавление элементов (без удаления)

```
let arr = [1, 4];
```

```
let removed = arr.splice(1, 0, 10, 20);
```

Расшифровка:

- начать с индекса 1
- удалить 0 элементов
- вставить 10 и 20

Результат:

- **arr** // [1, 10, 20, 4]
- **removed** // []

Пример 3. Замена элементов

```
let arr = [1, 2, 3];  
arr.splice(1, 1, 99);
```

Результат:

```
arr // [1, 99, 3]
```

Краткое сравнение

Метод	Что делает	Меняет массив
push	добавляет в конец	да
pop	удаляет с конца	да
unshift	добавляет в начало	да
shift	удаляет с начала	да
slice	копирует часть массива	нет
splice	изменяет массив	да

Вывод:

- Для добавления и удаления элементов следует использовать методы массива.
- `slice()` используется для копирования.
- `splice()` — самый универсальный, но изменяет массив.
- Методы делают код чище и безопаснее, чем работа с индексами.

6. Перебор массивов

Часто требуется последовательно обработать все элементы массива.

Для этого используются циклы.

Классический способ перебора массива:

```
let numbers = [10, 20, 30];
```

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

Особенности:

- используется индекс `i`;
- даёт полный контроль над перебором;
- можно изменять элементы массива.

Цикл for...of

Более современный и удобный способ:

```
for (let value of numbers) {  
    console.log(value);  
}
```

Особенности:

- перебирает значения, а не индексы;
- код короче и читабельнее;
- не подходит, если нужен индекс.

Вывод:

- Циклы позволяют обрабатывать все элементы массива.
- `for` — универсальный цикл с доступом к индексу.
- `for...of` — удобен для простого перебора значений.
- Важно правильно использовать `length`, чтобы избежать ошибок.

7. Методы перебора массивов

Методы перебора позволяют обрабатывать каждый элемент массива без явного использования циклов.

Код становится короче, понятнее и выразительнее.

forEach() — выполнение действия для каждого элемента
Синтаксис:

array.forEach(callback(currentValue, index, array), thisArg);

- callback — функция, которая будет вызвана для каждого элемента массива. Она принимает три аргумента:
- currentValue — текущий элемент массива.
- index (необязательно) — индекс текущего элемента.
- array (необязательно) — сам массив, который обрабатывается.
- thisArg (необязательно) — значение, которое будет использоваться как this внутри функции callback.

Пример 1:

```
numbers.forEach(function (item) {  
    console.log(item);  
});
```

Процесс выполнения:

- Метод `forEach` будет последовательно вызывать функцию для каждого элемента массива.
- Внутри функции `item` будет соответствовать текущему элементу массива.

Пример 2, со стрелочной функцией:

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

```
array.forEach((element) => {  
  console.log(element); // Вывод: 1, 2, 3  
});
```

Пример 3, использование индекса:

```
const fruits = ["apple", "banana", "orange"];

fruits.forEach((fruit, index) => {
  console.log(`${index}: ${fruit}`);
});
```

```
0: apple
1: banana
2: orange
```

map() — преобразование массива

Метод **map()** в JavaScript используется для создания нового массива, где каждый элемент является результатом выполнения функции для соответствующего элемента исходного массива.

Синтаксис:

```
const newArray = array.map(callback(currentValue, index, array),  
thisArg);
```

callback — функция, которая вызывается для каждого элемента массива. Она принимает три аргумента:

- **currentValue** — текущий элемент массива.
- **index** (необязательно) — индекс текущего элемента.
- **array** (необязательно) — сам массив, на котором вызывается **map**.
- **thisArg** (необязательно) — значение, используемое как **this** в функции **callback**.

Основные свойства `map()`:

- Создает новый массив: Метод `map()` не изменяет оригинальный массив, а возвращает новый массив с преобразованными значениями.
- Не выполняет побочные эффекты: `map()` предназначен для преобразования массива, а не для выполнения действий (например, вывода чего-либо в консоль). Для таких задач лучше использовать `forEach()`.
- Длина нового массива совпадает с длиной исходного массива: Каждый элемент нового массива соответствует элементу исходного массива, но может быть преобразован.

Пример 1. Преобразование массива (умножение элементов на 2):

```
const doubled = numbers.map((num) => num * 2);  
  
console.log(doubled); // [2, 4, 6, 8, 10]  
console.log(numbers); // [1, 2, 3, 4, 5]
```

Пример 2, преобразование строк в числа:

```
const stringNumbers = ["1", "2", "3", "4"];

const numbers = stringNumbers.map((str) => parseInt(str));

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

`filter()` — фильтрация массива

Метод `filter()` позволяет создать новый массив, в который попадут только те элементы исходного массива, которые прошли определенную «проверку».

Синтаксис:

```
const newArray = array.filter((element, index, array) => {  
  // Должен вернуть true, чтобы оставить элемент, или  
  false, чтобы отфильтровать  
});
```

- element: Текущий обрабатываемый элемент массива.
- index (необязательно): Индекс текущего элемента.
- array (необязательно): Сам исходный массив.

Главные правила:

- Не изменяет оригинал: `filter()` всегда возвращает новый массив. Исходный массив остается прежним (это принцип иммутабельности).
- Возвращает массив: Даже если условию соответствует всего один элемент, вы получите массив с одним элементом. Если ни один не подошел — пустой массив.
- Ожидает логическое значение: Колбэк-функция должна возвращать `true` или `false`.

Пример 1. Оставим только числа больше 10:

```
const numbers = [5, 12, 8, 130, 44];  
const filtered = numbers.filter(num => num > 10);  
  
console.log(filtered); // [12, 130, 44]
```

Пример 2. Оставим только четные числа:

```
let numbers = [1, 2, 3, 4, 5];  
let even = numbers.filter(n => n % 2 === 0);
```

`reduce()` — сведение массива к одному значению

Метод последовательно перебирает элементы и накапливает результат в специальную переменную — аккумулятор.

Синтаксис:

```
const result = array.reduce((accumulator, currentValue, index, array) => {  
  // Логика вычисления  
  return accumulator;  
}, initialValue);
```

- accumulator: Накопленный результат (то, что вернул прошлый шаг).
- currentValue: Текущий элемент массива.
- initialValue: Начальное значение аккумулятора (очень важный параметр!).

Пример 1, Сумма всех чисел:

```
const numbers = [10, 20, 30, 40];

const total = numbers.reduce((sum, current) => {
  |   return sum + current;
}, 0); // Начинаем с 0

console.log(total); // 100
```

Пример 2, Подсчет количества повторений:

```
const fruits = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple'];

const count = fruits.reduce((acc, fruit) => {
  acc[fruit] = (acc[fruit] || 0) + 1;
  return acc;
}, {});

console.log(count); // { apple: 3, banana: 2, orange: 1 }
```

Мы берем пустой объект {} и, проходя по массиву, добавляем в него названия фруктов как ключи. Если фрукт встретился первый раз, ставим значение 1, если второй — увеличиваем текущее число на 1.

Вывод:

- Методы перебора заменяют циклы в большинстве случаев.
- Они не изменяют исходный массив.
- Код с ними становится короче и читаемее.
- `reduce` — мощный инструмент, который можно изучать постепенно.

Вывод по теме лекции:

- Массивы используются для хранения и обработки списков данных.
- Основные операции: перебор, добавление, фильтрация, преобразование.
- Большинство задач решается с помощью методов `forEach`, `map`, `filter`, `reduce`.
- Практика работы с массивами необходима для дальнейшего изучения DOM и форм.

Контрольные вопросы:

- Как получить элемент массива по индексу?
- Почему индексация в массиве начинается с нуля?
- Как узнать количество элементов массива?
- Как изменить элемент массива по индексу?
- Как добавить элемент в конец или начало массива с помощью методов массива?
- Чем отличается метод `slice()` от `splice()`?
- Что возвращает метод `splice()`?
- Как удалить последний или первый элемент массива?
- Как с помощью метода `forEach()` перебрать все элементы массива?
- Чем отличается `map()` от `forEach()`?
- Как использовать `filter()` для создания нового массива из исходного?
- Как с помощью `reduce()` подсчитать сумму чисел в массиве?
- Что произойдёт, если обратиться к несуществующему индексу массива?
- Можно ли хранить элементы разных типов в одном массиве?
- Когда следует использовать методы перебора массива, а когда классический цикл `for`?

Домашнее задание:

<https://ru.hexlet.io/courses/js-basics>

хекслет колледж

@HEXLY.KZ