Массив

можно создать с двумя способами литеральный и нотационный

### 1. ****Литеральный способ (array literal notation)****

Это самый распространённый и предпочтительный способ создания массива:

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

### 2. ****Нотационный способ через конструктор (****new Array()****)****

let arr = new Array(1, 2, 3, 4);

Если создать таким спопсоб будет

let arr1 = new Array(5); // массив длиной 5, но без элементов (пустые слоты)

console.log(arr1); // [,,,,]

На самом деле в массиве **5 "пустых слотов" (empty slots)** — не undefined, не null, а именно **пустые ячейки**, которые **ещё не инициализированы**.

начает: «создать массив длиной 5, но без элементов внутри». Это разреженный массив (sparse array).

И когда ты выводишь его в консоль, она визуально показывает эти пустые слоты запятыми:

* Массив [ , , , , ] — это 5 пустых слотов → между ними 4 запятых.
* Количество запятых всегда на **один меньше количества слотов**.

**Слот** — это позиция в массиве.

* Если туда что-то записано — слот заполненный.
* Если не записано ничего — пустой слот, он даже не равен undefined, его просто **нет** в полном смысле.

Элементы массива нумеруются, начиная с нуля.

Мы можем получить элемент, указав его номер в квадратных скобках:

let fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits[0] ); // Яблоко

alert( fruits[1] ); // Апельсин

alert( fruits[2] ); // Слива

Мы можем заменить элемент:

fruits[2] = 'Груша'; // теперь ["Яблоко", "Апельсин", "Груша"]

…Или добавить новый к существующему массиву:

fruits[3] = 'Лимон'; // теперь ["Яблоко", "Апельсин", "Груша", "Лимон"]

Общее число элементов массива содержится в его свойстве length:

let fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits.length ); // 3

**В массиве могут храниться элементы любого типа.**

Например:

// разные типы значений

let arr = [ 'Яблоко', { name: 'Джон' }, true, function() { alert('привет'); } ];

// получить элемент с индексом 1 (объект) и затем показать его свойство

console.log( arr[1].name ); // Джон

// получить элемент с индексом 3 (функция) и выполнить её

arr[3](); // привет

Отличный вопрос! Да, **в массиве можно хранить все виды функций** в JavaScript:

1. **Function Declaration** (функция-объявление)
2. **Function Expression** (функция-выражение)
3. **Arrow Function** (стрелочная функция)

Но есть тонкости.

### 1. ****Function Declaration****

❌ Нельзя напрямую положить в массив  
Функция-декларация (объявление) **не возвращает значение**, её нельзя вставить внутрь массива **в месте, где ожидается выражение**.

Пример — так **нельзя**:

javascript

КопироватьРедактировать

let arr = [

function myFunc() { // это function expression, не declaration

return "Hello";

}

];

А это — **function expression**, хотя выглядит как объявление.

А вот **настоящее function declaration** вне массива:

function myFunc() {

return "Hello";

}

let arr = [myFunc]; // можно положить в массив по имени

### 2. ****Function Expression****

✅ Да, можно

javascript

КопироватьРедактировать

let arr = [

function() {

return "Anonymous Function";

},

function namedFunc() {

return "Named Function";

}

];

Обе — function expressions. Они помещаются в массив как значени

### 3. ****Arrow Function****

✅ Да, можно

javascript

КопироватьРедактировать

let arr = [

() => "Arrow 1",

(name) => `Hello, ${name}`

];

console.log(arr[0]()); // "Arrow 1"

console.log(arr[1]("Alice")); // "Hello, Alice"

**Function Declaration** — это инструкция, её нельзя вставлять внутрь выражения, как массив.

* **Function Expression** — создаёт функцию как значение, её можно положить в массив.
* Когда ты пишешь function sayHi() {} внутри массива — это **function expression**, даже если имя есть.

Хочешь использовать функции в массиве? Просто используй **function expressions** или **стрелочные функции**.

Если функция сразу вызыается нужно обернуть его в скобки

let arr = [

(function() {

console.log("Привет");

return "Готово!";

})()

];— это **анонимная функция, которая сразу же вызывается**.  
Такой приём называется **IIFE** (Immediately Invoked Function Expression — "немедленно вызываемое функциональное выражение"). Если было бы без обернутых скобок функция то была бы ошибка syntax error

**Получение последнего элемента в массиве**

**1. Через индекс:**

let arr = ["яблоко", "банан", "вишня"];

let last = arr[arr.length - 1];

console.log(last); // "вишня"

**2. Через метод .at(-1) (современный способ)**

let arr = ["яблоко", "банан", "вишня"];

let last = arr.at(-1);

console.log(last); // "вишня"

Если массив пустой

let arr = [];

let last = arr.at(-1); // undefined

// или

let last2 = arr[arr.length - 1]; // undefined

Оба способа просто вернут undefined, ошибки не будет.

Можно комбинировать также

let words = ["дом", "лес", "река"];

let lastWord = words.at(-1); // "река"

let lastChar = lastWord.at(-1); // "а"

console.log(lastChar); // "а"

## [Методы pop/push, shift/unshift](https://learn.javascript.ru/array" \l "metody-pop-push-shift-unshift)

Очередь – один из самых распространённых вариантов применения массива. В области компьютерных наук так называется упорядоченная коллекция элементов, поддерживающая два вида операций:

push добавляет элемент в конец.

shift удаляет элемент в начале, сдвигая очередь, так что второй элемент становится первым.

В программировании **очередь** (queue) — это **структура данных**, в которой элементы добавляются **в конец**, а извлекаются **с начала**. Массивы поддерживают обе операции.На практике необходимость в этом возникает очень часто. Например, очередь сообщений, которые надо показать на экране.

**Стек (stack)** — это структура данных, работающая по принципу **LIFO**: Last In, First Out (последним пришёл — первым ушёл).

## Представь стопку тарелок:

* Ты кладёшь новые тарелки **сверху**
* Забираешь тарелки **тоже сверху**

Вот это и есть **стек** — ты можешь взаимодействовать только с **верхушкой**.

let stack = [];

// Кладём в стек

stack.push("первый"); // теперь на дне

stack.push("второй"); // сверху

// Забираем со стека

console.log(stack.pop()); // "второй"

console.log(stack.pop()); // "первый"

## Методы, работающие с ****концом массива****

| Метод | Что делает | Пример |
| --- | --- | --- |
| push(x) | Добавляет элемент **в конец** | [1,2].push(3) → [1,2,3] |
| pop() | Удаляет элемент **с конца** | [1,2,3].pop() → 3, [1,2] |
| at(-1) | Возвращает **последний элемент** | [1,2,3].at(-1) → 3 |
| arr[arr.length - 1] | То же, что at(-1) вручную |  |
| arr.length-- | Удаляет **последний элемент** без возврата |  |

## Методы, работающие с ****началом массива****

| Метод | Что делает | Пример |
| --- | --- | --- |
| unshift(x) | Добавляет элемент **в начало** | [2,3].unshift(1) → [1,2,3] |
| shift() | Удаляет элемент **с начала** | [1,2,3].shift() → 1, [2,3] |
| at(0) | Возвращает **первый элемент** | [1,2,3].at(0) → 1 |
| arr[0] | То же самое вручную |  |

## [Внутреннее устройство массива](https://learn.javascript.ru/array" \l "vnutrennee-ustroystvo-massiva)

Массив это объект со специальным поведением.

Ключи, которые — **числа или строки, похожие на числа** ("0", "1", ..., "99999") — становятся **элементами массива**.

* Остальные ключи (например, "age") — это **обычные свойства объекта**.

let fruits = [];

fruits[99999] = "banana"; // это индекс ⇒ становится элементом массива

fruits.age = 25; // это обычное свойство ⇒ НЕ влияет на массив

свойство вроде fruits.age = 25 **сохраняется**, но **не как элемент массива**, а как **обычное свойство объекта**, которое:

* **не видно** при переборе массивных значений (for...of, .map(), forEach(), индексы);
* **не влияет** на length;
* **не участвует** в методах сортировки, фильтрации и других «чисто массивных» операциях.

console.log(Object.keys(fruits));

// ['0', 'age']

console.log(Object.getOwnPropertyNames(fruits));

// ['0', 'age', 'length']

Чем больше элементов содержит массив, тем больше времени потребуется для того, чтобы их переместить, больше операций с памятью.

То же самое происходит с unshift: чтобы добавить элемент в начало массива, нам нужно сначала сдвинуть существующие элементы вправо, увеличивая их индексы.

А что же с push/pop? Им не нужно ничего перемещать. Чтобы удалить элемент в конце массива, метод pop очищает индекс и уменьшает значение length.

Метод pop не требует перемещения, потому что остальные элементы остаются с теми же индексами. Именно поэтому он выполняется очень быстро.

Аналогично работает метод push.

**Перебор элементов**

**for – он перебирает вместе с индексом**

let arr = ["Яблоко", "Апельсин", "Груша"];

for (let i = 0; i < arr.length; i++) {

console.log( arr[i] );

}

Но для массивов возможен и другой вариант цикла, for..of:

let fruits = ["Яблоко", "Апельсин", "Слива"];

// проходит по значениям

for (let fruit of fruits) {

console.log( fruit );

}

Цикл for..of не предоставляет доступа к номеру текущего элемента, только к его значению, но в большинстве случаев этого достаточно. А также это короче.

Технически, так как массив является объектом, можно использовать и вариант for..in:

let arr = ["Яблоко", "Апельсин", "Груша"];

for (let key in arr) {

alert( arr[key] ); // Яблоко, Апельсин, Груша

}

### Почему не стоит использовать for...in с массивами:

#### 1. for...in ****перебирает**** все перечисляемые свойства объекта

Это означает, что он может перебирать **не только элементы массива**, но и **дополнительные свойства**, если они были добавлены вручную:

let arr = ["Яблоко", "Апельсин", "Груша"];

arr.test = "Привет";

for (let key in arr) {

console.log(key); // 0, 1, 2, test ❗

}

#### 2. ****Порядок итерации не гарантирован****

for...in **не гарантирует порядок** обхода ключей. Для массивов это может привести к неожиданному поведению, особенно если порядок важен.

**Length -** .length — это не "количество элементов", а "наибольший индекс + 1".

let arr = [];

arr[5] = "Привет";

console.log(arr.length); // 6

Массивы в JS индексируются с **нуля**.

* Ты **пропустил** индексы 0, 1, 2, 3, 4.
* Добавил элемент **в индекс 5**.
* Это значит, что **наибольший индекс** = 5.

length = (наибольший индекс) + 1

**Удаление не уменьшает length:**

let arr = ["Яблоко", "Апельсин", "Груша"];

delete arr[1]; // Удалим "Апельсин"

console.log(arr); // ["Яблоко", <пусто>, "Груша"]

console.log(arr.length); // 3 ❗

Индекс 1 удалён, но length остаётся 3.

А вот если изменить length вручную:

let arr = [1, 2, 3, 4, 5];

arr.length = 2;

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

Обрезка массива. Всё, что дальше второго элемента, удаляется.

В JavaScript **обращение к любой несуществующей ячейке массива по индексу** всегда возвращает undefined, даже если физически ячейка отсутствует.

Но это **не означает**, что ячейка существует с undefined внутри — она **вообще отсутствует**.

## [Многомерные массивы](https://learn.javascript.ru/array" \l "mnogomernye-massivy)

Массивы могут содержать элементы, которые тож е являются массивами. Это можно использовать для создания многомерных массивов, например, для хранения матриц:

toString - Массивы по-своему реализуют метод toString, который возвращает список элементов, разделённых запятыми.

let arr = [1, 2, 3];

alert( arr ); // 1,2,3

alert( String(arr) === '1,2,3' ); // true

а теперь так

alert( [] + 1 ); // "1"

alert( [1] + 1 ); // "11"

console.log( [1,2] + 1 ); // "1,21"

console.log( [1,2,3,4] + 1 ); //"1,2,3,41"

Массивы не имеют ни Symbol.toPrimitive, ни функционирующего valueOf, они реализуют только преобразование toString, таким образом, здесь [] становится пустой строкой, [1] становится "1", а [1,2] становится "1,2".

Когда бинарный оператор плюс "+" добавляет что-либо к строке, он тоже преобразует это в строку, таким образом

alert( "" + 1 ); // "1"

alert( "1" + 1 ); // "11"

alert( "1,2" + 1 ); // "1,21"

**Сравнения массива**

console.log([1, 2] == [1, 2]); // false

console.log([1, 2] === [1, 2]); // false

Потому что **в JavaScript массивы — это объекты**, а **объекты сравниваются по ссылке**, а не по содержимому.

let a = [1, 2];

let b = [1, 2];

let c = a;

console.log(a === b); // false — разные объекты в памяти

console.log(a === c); // true — одна и та же ссылка

Сравнение двух массивов через == или === вернёт true **только если они ссылаются на один и тот же объект** в памяти.

Чтобы сравнить нужно сравнивать **вручную**, поэлементно:

function arraysEqual(arr1, arr2) {

if (arr1.length !== arr2.length) return false;

for (let i = 0; i < arr1.length; i++) {

if (arr1[i] !== arr2[i]) return false;

}

return true;

}

console.log(arraysEqual([1, 2], [1, 2])); // true

В JavaScript сравнение **массивов** с **примитивами** (строкой, числом, булевым значением и т.п.) может вести себя неожиданно, потому что при таких сравнениях массив неявно преобразуется к примитиву.

[1,2] == "1,2" // true

[1] == 1 // true

[] == 0 // true

[] == false // true

[0] == false // true

[1,2] == "1,2"

[1,2].toString() → "1,2"

отличие splice от slice   
splice мутабельный для ориг массива а slice иммутабельный а также нельзя добавлять через slice элементы (push/shift)

**План урока**

1. Что такое массивы?
2. Создание массива.
3. Для чего они нужны и где они используются?
4. В чет отличия от объектов?
5. Методы массивов (добавление в середину)
6. Массивы могут хранить любые типы данные
7. Перебор элементов (виды переборов)
8. Задания

|  | ****Массив (****Array****)**** | ****Объект (****Object****)**** |
| --- | --- | --- |
| **Ключи** | Только **числовые индексы** (0, 1, 2...) | Любые **строки / символы** как ключи |
| **Используется** | Для **списков** значений (одинаковый тип) | Для **структур данных** (имя → значение) |
| **Перебор** | for, for...of, forEach, map | for...in, Object.keys(), Object.entries() |
| **Методы** | Есть **много методов**: push, pop, map, filter, reduce | Меньше встроенных методов |
| **Устройство** | Хранит значения **по индексам** | Хранит **пары ключ–значение** |

| Метод | Что делает | Пример |
| --- | --- | --- |
| push() | Добавляет в конец | arr.push(4) |
| pop() | Удаляет из конца | arr.pop() |
| unshift() | Добавляет в начало | arr.unshift(0) |
| shift() | Удаляет из начала | arr.shift() |
| splice() | Добавляет/удаляет/заменяет в середине | arr.splice(1, 2, "a") |
| reverse() | Переворачивает массив | arr.reverse() |
| sort() | Сортирует массив (по умолчанию как строки) | arr.sort() |
| fill() | Заполняет массив одним значением | arr.fill(0) |

| Метод | Что делает | Пример |
| --- | --- | --- |
| slice() | Копирует часть массива | arr.slice(1, 3) |
| concat() | Склеивает массивы | arr.concat([4, 5]) |
| map() | Преобразует каждый элемент | arr.map(x => x \* 2) |
| filter() | Фильтрует по условию | arr.filter(x => x > 0) |
| reduce() | Сводит массив к одному значению | arr.reduce((a, b) => a + b) |
| flat() | «расплющивает» вложенные массивы | arr.flat() |
| includes() | Проверяет наличие значения | arr.includes(3) |
| indexOf() | Индекс первого вхождения | arr.indexOf(2) |
| join() | Склеивает элементы в строку | arr.join(", ") |

| Метод | Что делает | Пример |
| --- | --- | --- |
| forEach() | Перебирает каждый элемент | arr.forEach(el => console.log(el)) |
| find() | Находит первый подходящий элемент | arr.find(x => x > 10) |
| findIndex() | Индекс первого подходящего элемента | arr.findIndex(x => x > 10) |
| every() | Проверяет, что **все** элементы соответствуют | arr.every(x => x > 0) |
| some() | Проверяет, что **хотя бы один** подходит | arr.some(x => x > 10) |