Массивы

Существует два варианта синтаксиса для создания пустого массива:

1. let arr = new Array();

У этого способа есть особенность. Если new Array вызывается с одним аргументом, который представляет собой число, он создаёт массив без элементов, но с заданной длиной.

```
let arr = new Array(2); // создастся ли массив [2]?
alert( arr[0] ); // undefined! нет элементов.
alert( arr.length ); // length 2
```

2. let arr = [];

Элементы массива нумеруются, начиная с нуля. Мы можем получить элемент, указав его номер в квадратных скобках. Мы можем заменить элемент или добавить новый к существующему массиву.

Общее число элементов массива содержится в его **свойстве** arr.length (без скобок):

```
let fruits = ["Яблоко", "Апельсин", "Слива"];
alert( fruits.length ); // 3
```

Свойство length автоматически обновляется при изменении массива. Если быть точными, это не количество элементов массива, а **наибольший цифровой индекс плюс один**. Если мы уменьшим его, массив станет короче. Таким образом, самый простой способ очистить массив – это arr.length = 0;.

Вывести массив целиком можно при помощи **alert**.

```
let fruits = ["Яблоко", "Апельсин", "Слива"];
alert( fruits ); // Яблоко, Апельсин, Слива
```

В массиве могут храниться элементы любого типа. Массив является объектом и, следовательно, ведёт себя как объект.Например, копируется по ссылке. Массив следует считать особой структурой, позволяющей работать с упорядоченными данными. Для этого массивы предоставляют специальные методы.

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
```

Очереди и стеки

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.

Существует и другой вариант применения для массивов – структура данных, называемая стек. Она поддерживает два вида операций:

- **push** добавляет элемент в конец.
- **pop** удаляет последний элемент.

Таким образом, новые элементы всегда добавляются или удаляются из «конца».Примером стека обычно служит колода карт: новые карты кладутся наверх и берутся тоже сверху.

Массивы в JavaScript могут работать и как очередь, и как стек. Мы можем добавлять/удалять элементы как в начало, так и в конец массива.В компьютерных науках структура данных, делающая это возможным, называется двусторонняя очередь.

Методы для добавления/удаления элементов:

arr.pop Удаляет последний элемент из массива и **возвращает его**:

```
let fruits = ["Яблоко", "Апельсин", "Груша"];
alert( fruits.pop() ); // удаляем "Груша" и выводим его
```

arr.push Добавляет элемент или элементы в конец массива и возвращает **новую, увеличенную длину** массива.:

```
let fruits = ["Яблоко", "Апельсин"];
fruits.push("Груша");
```

arr.shift Удаляет из массива первый элемент и **возвращает его**:

```
let fruits = ["Яблоко", "Апельсин", "Груша"];
alert( fruits.shift() ); // удаляем Яблоко и выводим его
```

arr.unshift Добавляет элемент или элементы в начало массива и возвращает **получившуюся длину**:

```
let fruits = ["Апельсин", "Груша"];
fruits.unshift('Яблоко');
```

arr.splice(pos, deleteCount, ...items) – начиная с индекса pos, удаляет deleteCount элементов и вставляет items. Возвращает массив из удалённых элементов. В этом и в других методах массива допускается использование отрицательного индекса. Он позволяет начать отсчёт элементов с конца. **Меняет исходный массив**.

```
let arr = ["Я", "изучаю", "JavaScript"];
// с позиции 2 удалить 0 элементов вставить "сложный", "язык"
arr.splice(2, 0, "сложный", "язык");
alert( arr ); // "Я", "изучаю", "сложный", "язык", "JavaScript"
```

arr.slice(start, end) – создаёт **новый массив**, копируя в него элементы с позиции start до end (**не включая end**). Оба индекса start и end могут быть отрицательными. В таком случае отсчёт будет осуществляться с конца массива.

```
let arr = ["t", "e", "s", "t"];
alert( arr.slice(-2) ); // s,t (копирует с -2 до конца)
```

let arr2 = arr.slice() – **копирует** весь массив.

arr.concat(...items) – возвращает **новый массив**: копирует все члены текущего массива и добавляет к нему items. Если какой-то из items является массивом, тогда берутся его элементы.

```
let arr = [1, 2];
// создать массив из: arr и [3,4], потом добавить значения 5 и 6
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

Перебор элементов

Одним из самых старых способов перебора элементов массива является цикл for по цифровым индексам:

```
let arr = ["Яблоко", "Апельсин", "Груша"];
for (let i = 0; i < arr.length; i++) {
  alert( 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) {
  alert( 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, но не следует.

arr.forEach(func) – вызывает func для каждого элемента. Ничего не возвращает. Результат функции (если она вообще что-то возвращает) отбрасывается и игнорируется.

```
arr.forEach(function(item, index, array) {
  // ... делать что-то с item
});
```

Например, этот код выведет на экран каждый элемент массива:

```
// Вызов alert для каждого элемента
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

Поиск в массиве

arr.indexOf/lastIndexOf(item, pos) – ищет item, начиная с позиции pos, и возвращает его **индекс** или **-1**, если ничего не найдено.

arr.includes(value) – возвращает **true**, если в массиве имеется элемент value, в противном случае false.

Методы используют строгое сравнение **===**. Отличием includes является то, что он правильно обрабатывает NaN в отличие от indexOf/lastIndexOf

```
const arr = [NaN];
alert( arr.indexOf(NaN) ); // -1 (должен быть 0, но === проверка на равенство не работает для NaN)
alert( arr.includes(NaN) );// true (верно)
```

arr.find/filter(func) – фильтрует элементы через функцию и отдаёт **первое/все** значения, при прохождении которых через функцию возвращается **true**.

arr.findIndex похож на find, но возвращает первый индекс вместо значения.

```
let result = arr.find(function(item, index, array) {
  // если true - возвращается текущий элемент и перебор прерывается
  // если все итерации оказались ложными, возвращается undefined
});

let results = arr.filter(function(item, index, array) {
  // если true - элемент добавляется к результату, и перебор продолжается
  // возвращается пустой массив в случае, если ничего не найдено
}); - возвращает массив из найденных элементов.

let users = [
  {id: 1, name: "Вася"},
  {id: 2, name: "Петя"},
  {id: 3, name: "Маша"}
];
// возвращает массив, состоящий из двух первых пользователей
let someUsers = users.filter(item => item.id < 3);
alert(someUsers.length); // 2
```

Преобразование массива

arr.map(func) – создаёт **новый массив** из результатов вызова func для каждого элемента.

```
let result = arr.map(function(item, index, array) {
  // возвращается новое значение вместо элемента
});
```

Например, здесь мы преобразуем каждый элемент в его длину:

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
alert(lengths); // 5,7,6
```

arr.sort(func) – сортирует массив «на месте», а потом возвращает его. **Меняет исходный массив.** Он возвращает отсортированный массив, но обычно возвращаемое значение игнорируется, так как изменяется сам arr.

По умолчанию элементы сортируются как строки. Чтобы использовать наш собственный порядок сортировки, нам нужно предоставить функцию в качестве аргумента arr.sort(). Функция должна для пары значений возвращать:

```
function compare(a, b) {
  if (a > b) return 1; // если первое значение больше второго
  if (a == b) return 0; // если равны
  if (a < b) return -1; // если первое значение меньше второго
}
Например, для сортировки чисел:
function compareNumeric(a, b) {
  if (a > b) return 1;
  if (a == b) return 0;
  if (a < b) return -1;
}
let arr = [ 1, 2, 15 ];
arr.sort(compareNumeric);
alert(arr);  // 1, 2, 15
```

На самом деле от функции сравнения требуется любое положительное число, чтобы сказать «больше», и отрицательное число, чтобы сказать «меньше».Это позволяет писать более короткие функции:

```
let arr = [ 1, 2, 15 ];
arr.sort( (a, b) => a - b );
alert(arr);  // 1, 2, 15
```

arr.reverse() – «на месте» меняет порядок следования элементов на противоположный и возвращает изменённый массив. **Меняет исходный массив**.

str.split(delim)/arr.join(glue) – преобразует строку в массив и обратно.

```
let str = "тест";
alert( str.split('') ); // т,е,с,т
let arr = ['Вася', 'Петя', 'Маша'];
let str = arr.join(';'); // объединить массив в строку через ;
alert( str ); // Вася;Петя;Маша
```

arr.reduce(func, initial) – вычисляет одно значение на основе всего массива, вызывая func для каждого элемента и передавая промежуточный результат между вызовами.

```
let value = arr.reduce(function(previousValue, item, index, array) {
  // ...
}, [initial]);
```

Функция применяется по очереди ко всем элементам массива и «переносит» свой результат на следующий вызов.Аргументы:

- *previousValue* – результат предыдущего вызова этой функции, равен initial при первом вызове (если передан initial),
- *item* – очередной элемент массива,
- *index* – его индекс,
- *array* – сам массив.

При вызове функции результат её вызова на предыдущем элементе массива передаётся как первый аргумент. Звучит сложновато, но всё становится проще, если думать о первом аргументе как «аккумулирующем» результат предыдущих вызовов функции. По окончании он становится результатом reduce.

Тут мы получим сумму всех элементов массива всего одной строкой:

```
let arr = [1, 2, 3, 4, 5];
let result = arr.reduce((sum, current) => sum + current, 0);
alert(result); // 15
```

Мы также можем опустить начальное значение. При отсутствии initial в качестве первого значения берётся **первый элемент массива**, а перебор стартует со **второго**. Но, если массив пуст, то вызов reduce без начального значения выдаст ошибку.

Метод arr.reduceRight работает аналогично, но проходит по массиву справа налево.

Другие методы

Array.isArray(arr) проверяет, является ли arr массивом.

```
alert(Array.isArray([])); // true
```

arr.some(fn)/arr.every(fn) проверяет массив. Функция fn вызывается для каждого элемента массива аналогично map. Если **какие-либо/все** результаты вызовов являются **true**, то метод возвращает true, иначе false.

arr.fill(value, start, end) – заполняет массив повторяющимися value, начиная с индекса start до end.

arr.copyWithin(target, start, end) – копирует свои элементы, начиная со start и заканчивая end, в собственную позицию target (перезаписывает существующие).

Array.from() – который принимает итерируемый объект или псевдомассив и делает из него «настоящий» Array. После этого мы уже можем использовать методы массивов.

```
const boxes = document.querySelectorAll(".box");
const boxesES6 = Array.from(boxes);
```

Для строки:

```
var string = "Hello";
const arrayFromString = Array.from(string);
console.log(arrayFromString);  // (5) ['H', 'e', 'l', 'l', 'o']
```

Документация по методу Array.from():
https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/from

Параметр thisArg

Почти все методы массива, которые вызывают функции – такие как find, filter, map, за исключением метода sort, принимают необязательный параметр **thisArg**. Значение параметра thisArg становится **this** для func.

Вот полный синтаксис этих методов:

```
arr.find(func, thisArg);
arr.filter(func, thisArg);
arr.map(func, thisArg);
```