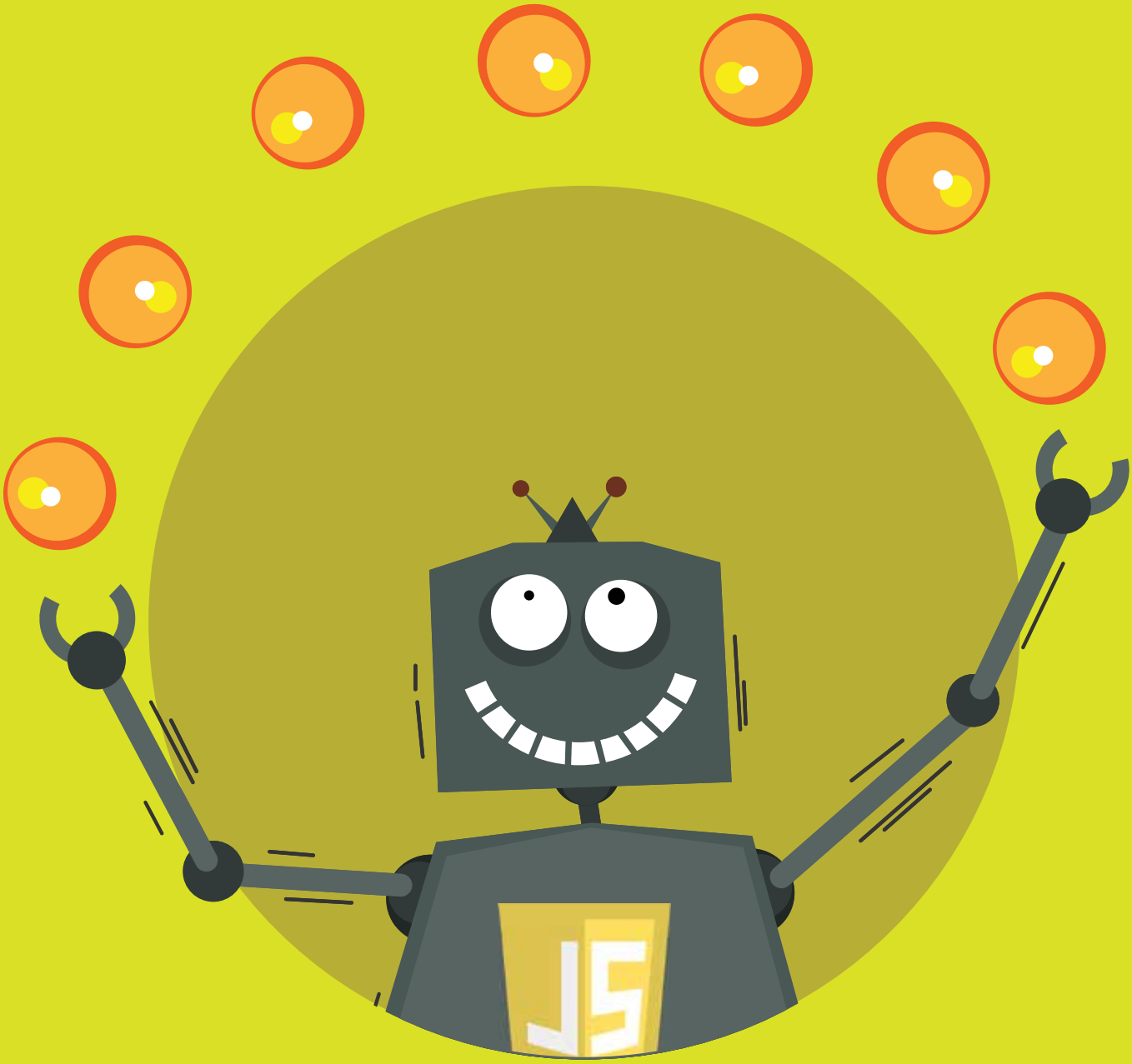


Объекты и массивы

LoftSchool
от мыслителя к создателю

Содержание

1

Объекты

С введением в объекты и массивы можно ознакомиться в методичке #1. Здесь, объекты и массивы будут рассмотрены более подробно.

Перечисление свойств объекта

Obejct.keys(obj)

Этот метод возвращает массив с именами перечисляемых свойств объекта:

```
var human = {
   name: 'Сергей',
   eyesColor: 'brown',
   hairColor: 'black'
};

console.log(Object.keys(human)); //выведет [ 'name',
'eyesColor', 'hairColor' ]
```

Object.getOwnPropertyNames(obj)

Этот метод возвращает массив с именами перечисляемых и неперечисляемых свойств объекта:

```
var human = {
   name: 'Сергей',
   eyesColor: 'brown',
   hairColor: 'black'
};

Object.defineProperty(human, 'old', {
   enumerable: false
});

human.old = 26;

console.log(Object.keys(human)); //выведет [ 'name',
'eyesColor', 'hairColor' ]

console.log(Object.getOwnPropertyNames(human)); //выведет
[ 'name', 'eyesColor', 'hairColor', 'old' ]
```

О том, что такое перечисляемые и не перечисляемые свойства, мы поговорим позднее.

Проверка наличия свойства

Существует несколько способов проверить наличие свойства в объекте:

```
var human = {
   name: 'Сергей',
   eyesColor: 'brown',
   hairColor: 'black'
};

console.log(typeof human.name != 'undefined'); //true
console.log('name' in human); //true
console.log(human.hasOwnProperty('name')); //true
```

Хотя с первым видом проверки надо быть осторожнее, т.к. может возникнуть такая ситуация:

```
var human = {
   name: 'Сергей',
   eyesColor: 'brown',
   hairColor: 'black',
   old: undefined
};

console.log(typeof human.old !== 'undefined'); //false
console.log('old' in human); //true
console.log(human.hasOwnProperty('old')); //true
```

Как видите, проверка на тип, при проверке на существование свойства, не всегда дает то, что нам нужно.

Передача по ссылке

В JavaScript простые типы (например, числа, строки и тд.) передаются по значению, а объекты и массивы - присваиваются и передаются по ссылке - это значит, что в переменной, в качестве значения, хранится не сам объект, а ссылка на то место в памяти, где хранится сам объект.

Чтобы это лучше понять - сравним обычные переменные и объекты:

```
var msg = "hello world!",  //значение хранится в
переменной
    msg2 = msg;  //msg2 получает копию значения
переменной msg
```

Во второй строке переменная, находящаяся в левой части оператора присваивания, получает копию значения переменной, находящейся в правой части этого оператора, т.е. теперь две переменные имеют каждая свое значение и изменив значение например в первой переменной, значение второй переменной останется неизменным.

Теперь объекты:

```
var user1 = {name: "Вася"}, //переменная содержит ссылку
на объект
    user2 = user1; //скопировали ссылку на объект
```

Здесь ситуация становится немного сложнее. При копировании переменной - копируется только ссылка, а объект остается в единственном экземпляре. Получается, что теперь две переменные ссылаются на один и тот же объект.

Следовательно, этим объектом можно оперировать с помощью переменной *user1* или *user2*. Так как объект всего один, то в какой бы переменной его не

изменяли — это отразится сразу на двух переменных:

```
var user1 = {name: "Вася"}, //переменная содержит ссылку
на объект
    user2 = user1; //скопировали ссылку на объект

user2.name = "Иван"; //изменили значение через user2

console.log(user1.name); //обратились к свойству через
user1. Полу Иван
```

Как видите, при присваивании объектов - копирования самого объекта не происходит. Вместо этого, вторая переменная так же начинает ссылаться на тот же объект.

С массивами - та же история:

```
var arr = [1, 2, 3, 4, 5, 6];

function func1(source) {
   source[1] = 100;
}

func1(arr);
console.log(arr); //выведет [ 1, 100, 3, 4, 5, 6 ]
```

Как видите - массив изменился не смотря на то, что напрямую мы его не редактировали, а использовали передачу массива в качестве параметра в функцию.

Метод

Метод - это функция, которая хранится в качестве значения в свойстве объекта и может вызываться посредством этого объекта.

```
var human = {
   name: 'Сергей',
   eyesColor: 'brown',
   hairColor: 'black',
   sayHello: function() {
       console.log("Привет!");
   }
};

human.sayHello(); //выведет Привет!
```

this

Метод должен иметь доступ к данным объекта для полноценной работы. Для доступа к объекту из метода используется ключевое слово ***this***. Оно ссылается на объект, в контексте которого вызван метод и позволяет обращаться к другим его методам и свойствам:

```
var human = {
   name: 'Сергей',
   eyesColor: 'brown',
   hairColor: 'black',
   sayHello: function() {
       console.log("Привет! Меня зовут", this.name);
   }
};

human.sayHello(); //выведет Привет! Меня зовут Сергей
```

this доступно только внутри методов и указывает на объект, из которого этот метод вызвали.

Вот еще пример:

```
var human = {
   name: 'Сергей',
   eyesColor: 'brown',
   hairColor: 'black',
   sayHello: function() {
       console.log("Привет! Меня зовут", this.name);
   }
};

var human2 = {
   name: 'Света',
   eyesColor: 'blue',
   hairColor: 'black',
};

human2.sayHello = human.sayHello;
human2.sayHello(); //выведет Привет! Меня зовут Света
```

Обратите внимание, что у объекта *human2* нет метода sayHello. Вместо этого, мы сами добавляем в него метод sayHello и вызываем его. Соответственно, **this** будет указывать на *human2*, т.к. sayHello вызван из объекта *human2*.

На самом деле, **this** указывает на контекст функции, а не на объект, из которого вызывается метод. Но об этом мы будем говорить на одном из следующих занятий, поэтому пока, чтобы не забивать себе голову, можете думать, что **this** внутри метода - это объект, из которого вызывается метода.

Создание свойств (продвинутый уровень)

При создании свойства объекта, можно более детально описывать это свойство - установив дополнительные параметры. В таком случае, свойство необходимо создавать через метод ***Object.defineProperty***.

Первым параметром, метод принимает объект, в котором необходимо создать свойство, вторым - имя свойства, которое необходимо создать и третьим - параметры для создаваемого свойства.

Вот параметры свойства, которые мы можем указать:

```
enumerable
```

Если указать true, то это свойство будет "перечисляемым". То есть оно будет доступно при переборе объекта оператором **for..in** и при помощи метода **Object.keys()**

При создании свойства обычным способом - равно **true**.
При создании свойства через Object.defineProperty - равно **false**.

value
Значение, которое будет присвоено свойству. Может быть любым допустимым значением JavaScript (числом, объектом, функцией и т.д.).
Значение по умолчанию - undefined.

writable
Если указать **true**, то значение свойста может быть с любой момент изменено с помощью оператора присваивания.
При создании свойства обычным способом - равно **true**.
При создании свойства через **Object.defineProperty** - равно **false**.

get

Функция, которая будет автоматически вызвана, при попытке обратиться к свойству. Функция должна вернуть какое-то значение.

set

Функция, которая будет автоматически вызвана, при попытке записать в свойство новое значение. В качестве параметра будет передано значение, которое должно быть установлено.

configurable

Если указать **true**, то свойство можно будет изменять через **defineProperty** и удалять при помощи оператора **delete**.
При создании свойства обычным способом - равно **true**.
При создании свойства через **Object.defineProperty** - равно **false**.

Посмотрите на пример, в котором собраны некоторые из этих параметров:

```
var human = {
    name: 'Сергей',
    lastName: 'Мелюков',
};

Object.defineProperty(human, 'fullName', {
    get: function() {
        return this.name + ' ' + this.lastName;
    },
    set: function(newName) {
        var newData = newName.split(' ');

        if (!newData[0] || !newData[1]) {
            throw new Error('имя указано неверно!');
        }
```

```
        this.name = newData[0];
        this.lastName = newData[1];
    }
});

console.log(human.fullName); //будет вызван метод get,
указанный при создании свойств fullName

//ниже будут выведены только свойства name и lastName
//fullName выведено не будет, т.к. параметр этог
свойства, по умолчанию, равен false

for (var prop in human) {
    console.log(prop);
}

console.log(Object.keys(human)); //выведет [ 'name',
'lastName' ] т.к. keys() возвращает только перечисляемые
свойства

console.log(Object.getOwnPropertyNames(human)); //выведет
[ 'name', 'lastName', 'fullName' ] т.к. keys() возвращает
все виды свойств

human.fullName = 'Дмитрий Ковальчук'; //будет вызван
метод set, указанный при создании свойств fullName

console.log(human.name); //выведет Дмитрий
console.log(human.lastName); //выведет Ковальчук

human.fullName = 'Дима'; //выбросится исключение, т.к.
имя указано в неверном формате
```

2

Массивы

Усечение и увеличение массива

При работе с массивами, длина массива(***свойства length***) автоматически обновляется, поэтому нам самим не приходится об этом заботиться. Но стоит упомянуть об одной детали - свойство **length** доступно не только для чтения, но и для записи. Если свойству **length** указать значение меньше текущего, то массив укорачивается до заданной длины.

Любые элементы, не попадающие в новый диапазон индексов - отбрасываются и их значения теряются, даже если потом вернуть **length** обратно - значения не будут восстановлены.

```
var arr = [5, 2, 4, 9];

arr.length = 1; //укорачиваем массив до 1 элемента
console.log(arr); //выведет [5]
arr.length = 4; //восстановим прежнее количество
элементов
console.log(arr); //выведет [5, undefined, undefined,
undefined]
```

Самым простым способом очистить массив будет: **foo.length = 0.**
Если свойство **length** сделать большим, чем его текущее значение, в конец массива добавятся новые неопределенные элементы, увеличивая массив до указанного размера. Это можно наблюдать в примере выше.

Методы

pop/push

Метод push() добавляет один или несколько новых элементов в конец массива и возвращает его новую длину. ***Метод pop()*** - удаляет последний

элемент массива, уменьшает длину массива и возвращает удаленное им значение. Стоит обратить внимание на то, что оба эти метода изменяют массив, а не создают его модифицированную копию.

```
var arr = [];

arr.push(1,2);   // foo: [1,2] Возвращает 2
arr.pop();       // foo: [1] Возвращает 2
arr.push(3);     // foo: [1,3] Возвращает 2
arr.pop();       // foo: [1] Возвращает 3
arr.push([4,5]); // foo: [1,[4,5]] Возвращает 2
arr.pop()        // foo: [1] Возвращает [4,5]
arr.pop();       // foo: [] Возвращает 1
```

shift/unshift

Методы shift() и unshift() ведут себя во многом также, как **pop()** и **push()**, за исключением того, что они вставляют и удаляют элементы в начале массива. Метод **unshift()** добавляет один или несколько элементов в начало массива и возвращает новую длину массива. Метод **shift()** удаляет первый элемент массива и возвращает его значение, смещая все последующие элементы для занятия свободного места в начале массива.

```
var arr = [];

arr.unshift(1);          // f:[1] Возвращает: 1
arr.unshift(22);         // f:[22,1] Возвращает: 2
arr.shift();             // f:[1] Возвращает: 22
arr.unshift(3, [4, 5]); // f:[3,[4,5],1] Возвращает: 3
arr.shift();             // f:[[4,5],1] Возвращает: 3
arr.shift();             // f:[1] Возвращает: [4,5]
arr.shift();             // f:[] Возвращает: 1
```

join

Метод Array.join() используется для объединения элементов массива в одну строку. Методу можно передать необязательный строковой аргумент, который будет использоваться для разделения элементов в строке. Если разделитель не задан, то при вызове метода символом-разделителем по умолчанию будет запятая.

```
var arr = ["Ветер", "Дождь", "Огонь"],
   joined = arr.join();

console.log(joined); //выведет Ветер,Дождь,Огонь
```

Метод Array.join() является обратным по отношению к методу **String.split()**, который создает массив путем разбиения строки на фрагменты.

```
var source = "Ветер,Дождь,Огонь",
   arr = source.split(',');

console.log(arr); //выведет [ 'Ветер', 'Дождь', 'Огонь' ]
```

reverse

Метод Array.reverse() меняет порядок следования элементов в массиве на противоположный и возвращает массив с переставленными элементами. Этот метод не создает новый массив с переупорядоченными элементами, а переупорядочивает их в уже существующем массиве.

```
var arr = ['Ветер', 'Дождь', 'Огонь'];

arr.reverse();
console.log(arr); //выведет [ 'Огонь', 'Дождь', 'Ветер' ]
```

concat

Метод Array.concat() создает и возвращает новый массив, содержащий элементы исходного массива, для которого был вызван метод **concat()**, последовательно дополненный значениями всех аргументов, переданных методу **concat()**. Если какой-либо из этих аргументов сам является массивом, тогда будут добавлены все его элементы. Имена массивов используются в качестве аргументов и указываются в том порядке, в котором нужно объединить их элементы. Не изменяет исходных массив.

```
var arr = [1, 2, 3];

arr.concat(4, 5)           //возвращает [1,2,3,4,5]
arr.concat([4, 5]);        //то же самое – возвращает
[1,2,3,4,5]
arr.concat([4, 5], [6, 7]); //возвращает [1,2,3,4,5,6,7]
```

sort

Метод Array.sort() сортирует элементы массива и возвращает отсортированный массив. Если метод **sort()** вызывается без аргумента, то он сортирует элементы массива в алфавитном порядке (временно преобразует их в строки для выполнения сравнения). Метод **sort** изменяет массив.

```
var arr = ["Киви", "Апельсины", "Груши"];

console.log(arr.sort()); //выведет Апельсины, Груши, Киви
```

Как было сказано ранее - **sort** сортирует значения как строки. Из-за этого, числа становится не в порядке убывания, а в алфавитном порядке:

```
var arr = [10, 2, 5, 1];

arr.sort();

console.log(arr); //выведет [ 1, 10, 2, 5 ]
```

Обратите внимание, что 10 явно находится не на своем месте.

Для сортировки в каком-либо ином порядке, отличном от алфавитного, можно передать методу **sort()** в качестве аргумента функцию сравнения. Следует однако учесть, что функцию сравнения придется писать самим. Эта функция должна иметь два параметра, так как она устанавливает, какой из двух ее аргументов должен присутствовать раньше в отсортированном списке:

```
function mySort(a, b) {
   if (a < b) {
       return -1;
   } else if (a > b) {
       return 1;
   }

   return 0;  //если a == b
}

var arr = [10, 2, 5, 1];

arr.sort(mySort);  //в качестве аргумента передается
только имя функции

console.log(arr); //выведет [ 1, 2, 5, 10 ]
```

slice

Метод Array.slice() используется для копирования указанного участка из массива и возвращает новый массив содержащий скопированные элементы. Исходный массив при этом не меняется.

Метод принимает два аргумента, которые определяют начало и конец возвращаемого участка массива. Метод копирует участок массива, начиная от **begin** до **end**, не включая **end**. Если указан только один аргумент, возвращаемый массив будет содержать все элементы от указанной позиции до конца массива. Можно использовать отрицательные индексы - они отсчитываются с конца массива.

```
var arr = [1,2,3,4,5];

arr.slice(0,3);   //возвращает [1,2,3]
arr.slice(3);    //возвращает [4,5]
arr.slice(1,-1);  //возвращает [2,3,4]
arr.slice(-3,-2); //возвращает [3]
```

splice

Метод Array.splice() - это универсальный метод для работы с массивами. Он изменяет массив, а не возвращает новый измененный массив, как это делают, например, методы **slice()** и **concat()**. Метод **splice** может удалять элементы из массива, вставлять новые элементы, заменять элементы. Он возвращает массив, состоящий из удаленных элементов, если ни один из элементов не был удален, вернет пустой массив. Первый аргумент указывает индекс в массиве, с которого начинается вставка или удаление элементов. Второй аргумент задает количество элементов, которые должны быть удалены из массива начиная с индекса, указанного в первом аргументе, если второй аргумент равен 0, то элементы не будут удалены. Если второй аргумент опущен, удаляются все элементы массива начиная с указанного индекса до конца массива. При использовании отрицательного номера позиции, отсчет элементов будет с конца массива.

После второго аргумента можно, через запятую, указать значения, которые будут вставлены, начиная с индекса, указанного первым аргументом. Посмотрите на примеры для лучшего понимания.

Удаление элементов:

```
var fruits = ["апельсины", "яблоки", "груши",
"виноград"],
      deleted = fruits.splice(2, 2);

console.log(deleted); //выведет ["груши", "виноград"]
console.log(fruits); //выведет ["апельсины", "яблоки"]
```

Удаление элементов и вставка новых:

```
var fruits = ["апельсины", "яблоки", "груши",
"виноград"],
      deleted = fruits.splice(2, 2, "киви", "дыня");

console.log(deleted); //выведет ["груши", "виноград"]
console.log(fruits); //выведет ["апельсины", "яблоки",
"киви", "дыня"]
```

Вставка новых элементов без удаления:

```
var fruits = ["апельсины", "яблоки", "груши",
"виноград"],
      deleted = fruits.splice(2, 0, "киви", "дыня");

console.log(deleted); //выведет []
console.log(fruits); //выведет ["апельсины", "яблоки",
"киви", "дыня", "груши", "виноград"]
```

indexOf и lastIndexOf

Метод indexOf возвращает индекс элемента, значение которого равно значению, переданному методу в качестве аргумента.

Если одинаковых вхождений несколько, то выбирается первый индекс. Если элемент с искомым значением не найден, то метод вернет **-1**. Внутри метода для поиска используется строгое сравнение **(===)**:

```
var arr = [1,2,3,3,4,5,3];

arr.indexOf(1);   //вернет 0
arr.indexOf(3);   //вернет 2
arr.indexOf('3'); //вернет -1
arr.indexOf(3,4); //вернет 6
arr.indexOf(35);  //вернет -1
arr.indexOf(2);   //вернет 1
```

Метод lastIndexOf() тоже возвращает индекс элемента, значение которого равно значению, переданному методу в качестве аргумента. Разница лишь в том, что метод **lastIndexOf()** выбирает индекс последнего вхождения.

```
var arr = [1,2,3,3,4,5,2,3];

arr.lastIndexOf(3);  //вернет 7
arr.lastIndexOf(35); //вернет -1
arr.lastIndexOf(2);  //вернет 6
```

Методы итераторы

forEach

Синтаксис метода:

```
имя_массива.forEach(callback, thisArg)
```

В качестве первого аргумента указывается callback-функция, которую метод forEach() будет вызывать для каждого элемента массива. Реализацию вызываемой функции-обработчика нужно писать самим. В вызываемую функцию будет автоматически передано 3 аргумента: первый - значение текущего элемента массива, второй - индекс текущего элемента массива, и третий - сам массив. Однако, если нужно использовать только значения элементов массива, можно написать функцию только с одним параметром. Второй аргумент - thisArg (необязательный). При помощи него можно указать - на что будет указывать this внутри callback-функци

```
var arr = [2, 3, 4];

function foo(value) {
   var sum = value * this;

   console.log(sum);
}

arr.forEach(foo, 5); //второй аргумент будет передан в
качестве значения this
```

filter

Синтаксис метода:

```
имя_массива.filter(callback, thisObject)
```

Метод filter() создает и возвращает новый массив, который будет содержать только те элементы массива, для которых вызов функции callback возвратит true.

```
function isBig(element, index, array) {  //возвращает
числа, которые больше или равны 10

   return (element >= 10); //если значение элемента
больше или равно 10 – выражение вернет true
}

var filtered = [11, 3, 7, 50, 25].filter(isBig);
//filtered[11, 50, 25]
```

map

Метод map() создает и возвращает новый массив, который будет состоять из результатов вызова функции **callback(item, idx, ar)** для каждого элемента массива.

```
var a = [5, 6, 7],
    b = a.map(function(item, idx, arr) {
          return item * item;
      }); // b = [25, 36, 49]
```

every и some

Метод every() возвращает **true**, если для всех элементов массива указанная функция, используемая для их проверки, вернет **true**.

Метод some() возвращает **true**, если проверяющая функция вернула **true** хотя бы для одного элемента:

```
var a = [1, 2, 3, 4, 5],
    b = [1, 2, 3, 14, 5];

a.every(function(x) {
   return x < 10;
}); //true, т.к. все значения < 10.

b.some(function(x) {
   return x > 10;
}); //true т.к одно из значений > 10
```

reduce и reduceRight

Синтаксис методов:

```
имя_массива.reduce(callback, initialValue)
имя_массива.reduceRight(callback, initialValue)
```

Метод reduce() применяет указанную функцию **(callback)** в отношении сразу двух значений в массиве, перебирая элементы слева направо, сохраняя при этом промежуточный результат.

Аргументы функции callback: (previousValue, currentItem, index, array)

previousValue	возвращаемый результат callback функции (он же промежуточный результат)
currentItem	текущий элемент массива (элементы перебираются по очереди слева-направо)
index	индекс текущего элемента
array	обрабатываемый массив

initialValue (инициализирующее значение) - объект, используемый в качестве первого аргумента первого вызова функции **callback**. Проще говоря, значение previousValue при первом вызове равно **initialValue**. Если **initialValue** нет, то оно равно первому элементу массива, а перебор начинается со второго:

```
var a = [1, 2, 3, 4, 5],
      result;

function foo(prevSum, curNum) {
   return prevSum + curNum;
}

result = a.reduce(foo, 0);
console.log(result); //15
```

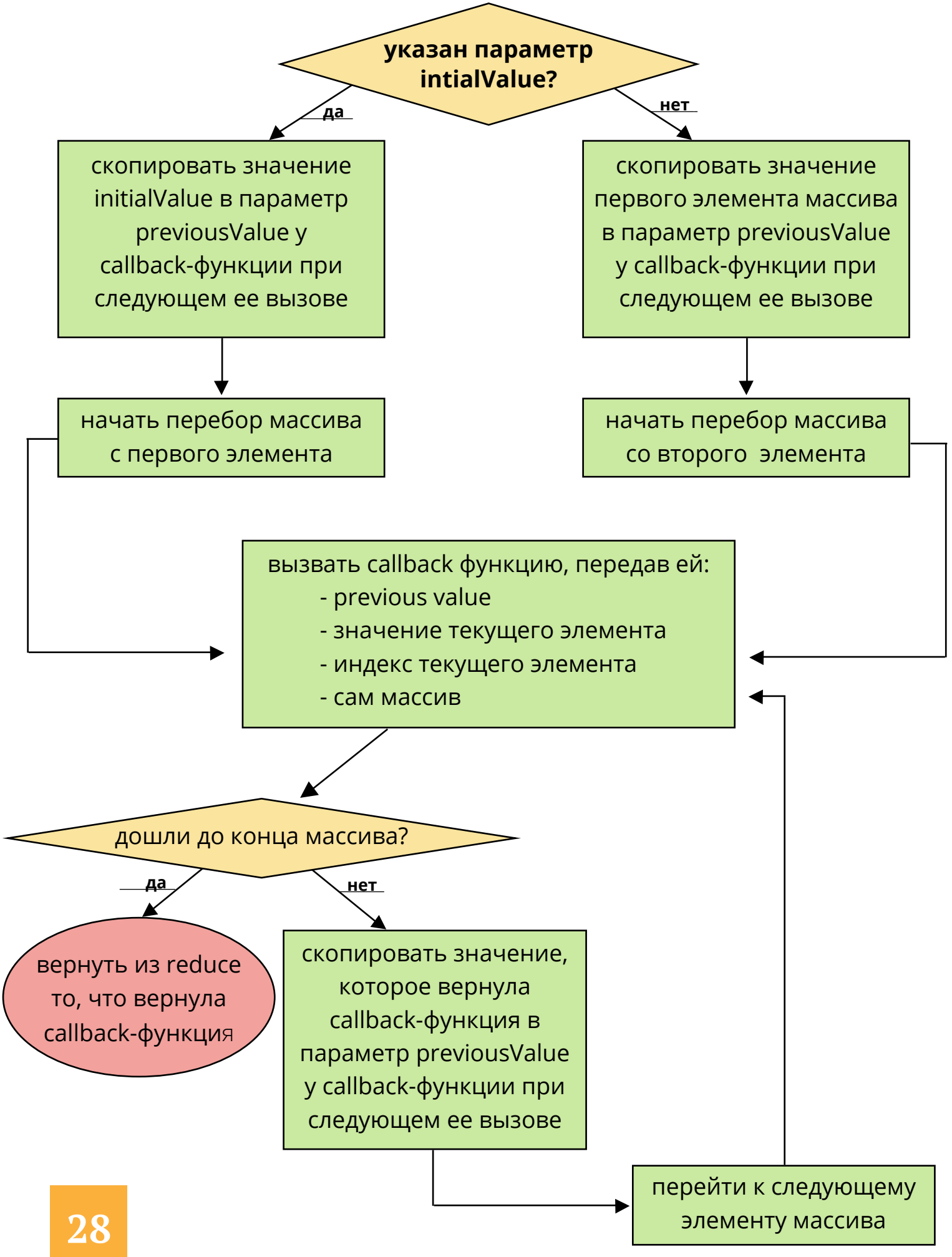
Метод reduceRight работает аналогично методу **reduce**, но идет по массиву справа-налево:

```
var arr = ["h","o","m","e"];

function rev(prevStr, curItem) {
   return prevStr + curItem;
}

console.log(arr.reduceRight(rev)); //emoh
```

В виде блок-схемы, работа метода reduce будет выглядеть так:


указан параметр intialValue?
да
нет
скопировать значение initialValue в параметр previousValue у callback-функции при следующем ее вызове
скопировать значение первого элемента массива в параметр previousValue у callback-функции при следующем ее вызове
начать перебор массива с первого элемента
начать перебор массива со второго элемента
вызвать callback функцию, передав ей:
- previous value
- значение текущего элемента
- индекс текущего элемента
- сам массив
дошли до конца массива?
да
нет
вернуть из reduce то, что вернула callback-функция
скопировать значение, которое вернула callback-функция в параметр previousValue у callback-функции при следующем ее вызове
перейти к следующему элементу массива