Продвинутый JS I. Замыкания. Методы массивов с функциями обратного вызова

Мікалай Янкойць

Адукар

Повестка дня

1. Функции высшего порядка
2. Замыкания
3. Анонимные функции
4. Методы массивов с обратными вызовами

Функция как значение

Функция – это объект. Это значит, что её, как и любые другие значения JS, можно передавать в качестве аргумента в другие функции, а также возвращать из них через return.

```
function double(x) {
    return x * 2;
}
function decrement(x) {
    return x - 1;
}
function doFuncWithX(func, x) {
    return func(x);
}
console.log(doFuncWithX(double, 3)); // 6
console.log(doFuncWithX(decrement, 1)); // 0
```

```
function createFunc() {
    let cube = function(x) {
        return x*x*x;
    }
    return cube;
}

let powerOfThree = createFunc();
console.log(powerOfThree);
// ƒ (x) { return x*x*x; }
console.log(powerOfThree(2)); // 8
```

Функция высшего порядка

Функция, которая принимает в качестве аргументов другие функции или возвращает другую функцию в качестве результата, называется **функцией высшего порядка**.

Например, функции doFuncWithX и createFunc с прошлого слайда – функции высшего порядка.

Функции, которые передают в другие функции в качестве аргумента, называют **функциями обратного вызова (callback function)**.

Немного повторения

- Код всегда выполняется в некотором **контексте** – окружении, хранящем все параметры, необходимые для правильной работы программы.
- **Каждый** вызов функции создаёт специальный **локальный** контекст, в котором хранятся все доступные переменные, аргументы функции, **области видимости** и многое другое.
- Код вне функций выполняется в **глобальном** контексте (единственном!)
- При каждом вызове функции её контекст выполнения (только что созданный!) сохраняется в специальной структуре – **стеке вызовов**.

Лексическое окружение

В контексте выполнения хранится куча полезных данных, в том числе **лексическое окружение** – объект, в котором хранятся все доступные переменные и их значения.

Когда мы вызываем функцию, мы тем самым создаём для неё контекст выполнения. А значит, и её собственное лексическое окружение.

Как выполняется код: пример

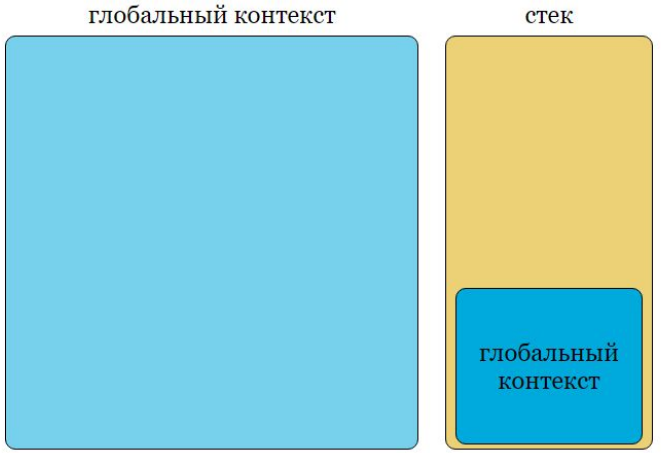
На примере этой простой функции попробуем (с некоторыми упрощениями) проследить, как выглядят лексические окружения на каждом этапе выполнения программы.

```
1  let a = 5;
2  function double(x) {
3    let result = x * 2;
4    return result;
5  }
6  let b = double(a);
7  console.log("выводим результат: " + b);
```

Как выполняется код: строка 0, ч.1

В момент запуска скрипта интерпретатор создаёт глобальный контекст выполнения и отправляет его на вершину стека вызовов.
(Голубой квадрат – лексическое окружение этого контекста.)

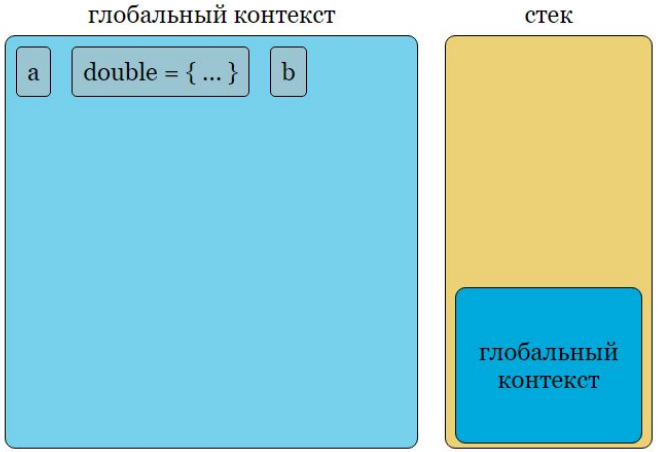
```
   /* мы здесь */
1  let a = 5;
2  function double(x) {
3    let result = x * 2;
4    return result;
5  }
6  let b = double(a);
7  console.log("выводим результат: " + b);
```



Как выполняется код: строка 0, часть 2

За счёт всплытия (hoisting) идентификаторы a, b и double записываются в лексическое окружение ещё до того, как начнёт выполняться код. double сразу получает значение (код функции).

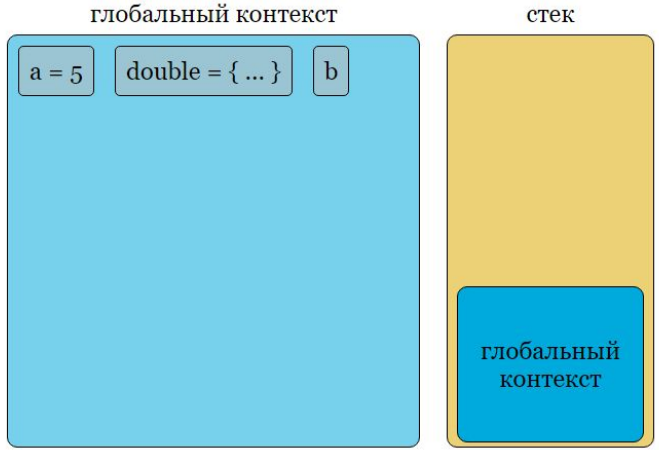
```
   /* мы всё ещё здесь */
1  let a = 5;
2  function double(x) {
3    let result = x * 2;
4    return result;
5  }
6  let b = double(a);
7  console.log("выводим результат: " + b);
```



Как выполняется код: строки 1-5

На первой строке скрипта переменная a получает значение 5 (и выходит из временной мёртвой зоны). Строки 2-5 игнорируются, пока функция double не будет вызвана.

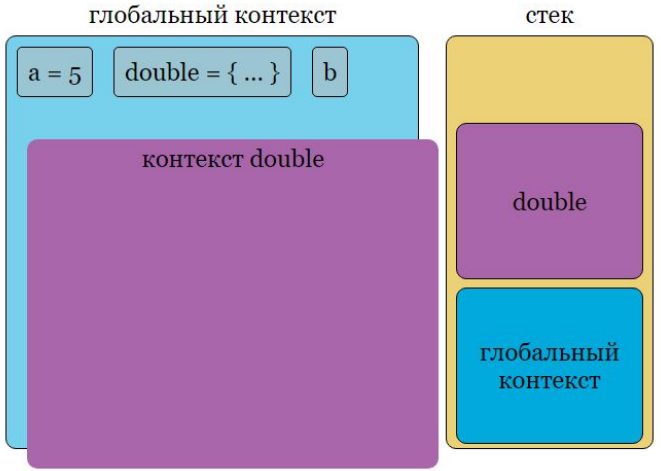
```
1  let a = 5; /* мы здесь */
2  function double(x) {
3    let result = x * 2;
4    return result;
5  }
6  let b = double(a);
7  console.log("выводим результат: " + b);
```



Как выполняется код: строка 6, часть 1

Чтобы присвоить значение переменной b, интерпретатор вызывает функцию double. Создаётся локальный контекст, он помещается на вершину стека вызовов.

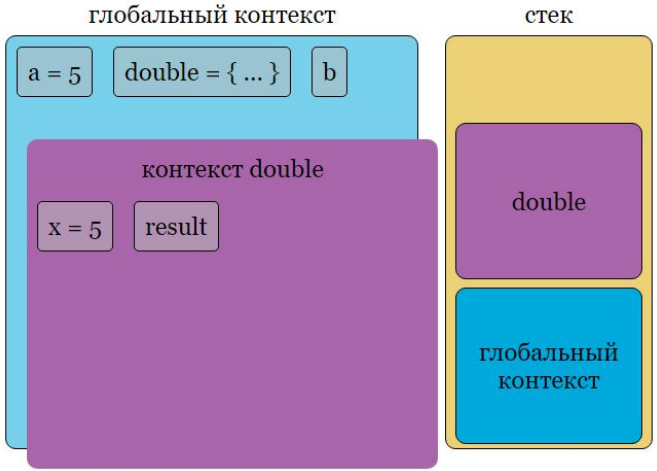
```
1  let a = 5;
2  function double(x) { /* идём сюда */
3    let result = x * 2;
4    return result;
5  }
6  let b = double(a); /* мы здесь */
7  console.log("выводим результат: " + b);
```



Как выполняется код: функция / строка 2

Перед тем, как начать выполнять код функции, интерпретатор записывает в лексическое окружение double аргумент x (со значением 5) и (за счёт всплытия) переменную result.

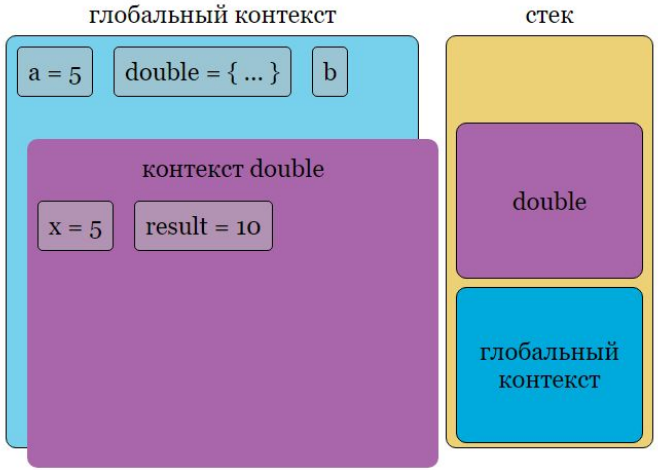
```
1  let a = 5;
2  function double(x) { /* мы здесь */
3    let result = x * 2;
4    return result;
5  }
6  let b = double(a);
7  console.log("выводим результат: " + b);
```



Как выполняется код: функция / строка 3

Интерпретатор подсчитывает x * 2 и присваивает результат переменной result.

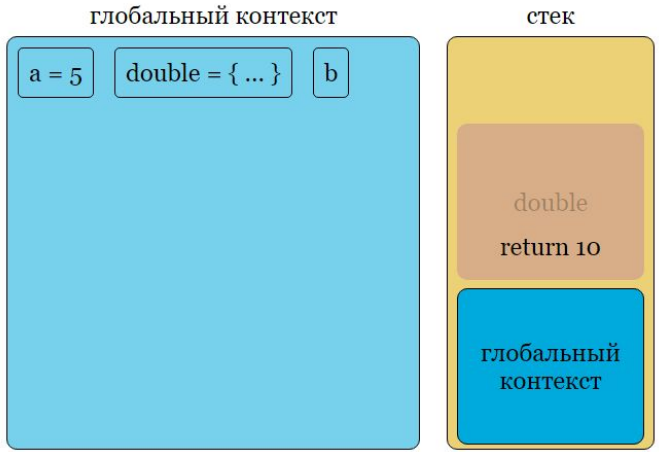
```
1  let a = 5;
2  function double(x) {
3    let result = x * 2; /* мы здесь */
4    return result;
5  }
6  let b = double(a);
7  console.log("выводим результат: " + b);
```



Как выполняется код: функция / строка 4

Интерпретатор встречает return, «запоминает» значение result, «снимает» локальный контекст функции double с вершины стека, а потом удаляет его вместе с его лексическим окружением.

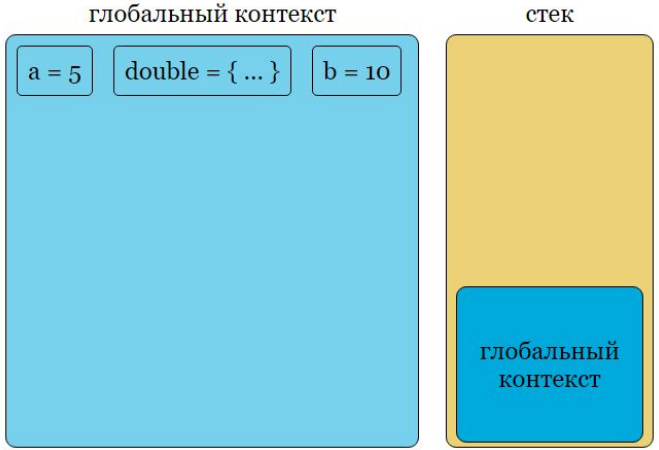
```
1  let a = 5;
2  function double(x) {
3    let result = x * 2;
4    return result; /* мы здесь */
5  }
6  let b = double(a); /* сейчас вернёмся сюда */
7  console.log("выводим результат: " + b);
```



Как выполняется код: строка 6, ч.2 + строка 7

Переменной b присваивается значение 10, возвращённое из функции. В строке 7 значение b конкатенируется со строкой и выводится в консоль.

```
1  let a = 5;
2  function double(x) {
3    let result = x * 2;
4    return result;
5  }
6  let b = double(a); /* мы снова здесь */
7  console.log("выводим результат: " + b);
```



Как выполняется код: пример 2

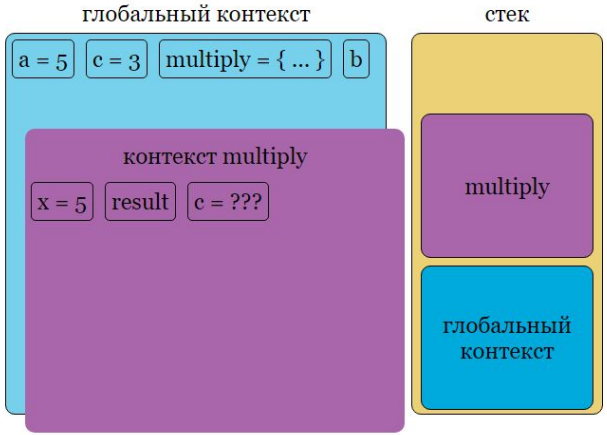
Усложним исходный код и проследим шаги, на которых поведение интерпретатора меняется.

```
1  let a = 5;
2  let c = 3;
3  function multiply(x) {
4    let result = x * c;
5    return result;
6  }
7  let b = multiply(a);
8  console.log("выводим результат: " + b);
```

Как выполняется код: строка 4, ч.1

Инициализация и начало скрипта проходят аналогично. Но после вызова функции multiply, на строке 4 интерпретатор сталкивается с проблемой: ему нужно значение переменной c, которая не описана в лексическом окружении multiply.

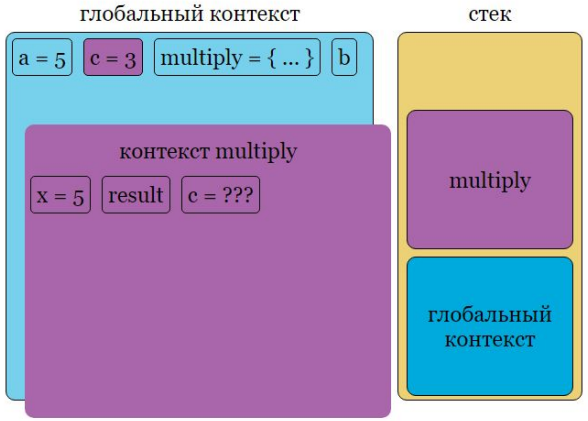
```
1  let a = 5;
2  let c = 3;
3  function multiply(x) {
4    let result = x * c; /* мы здесь */
5    return result;
6  }
7  let b = multiply(a);
8  console.log("выводим результат: " + b);
```



Как выполняется код: строка 4, ч.2

Не найдя переменную в лексическом окружении локального контекста multiply, интерпретатор «стучится» в родительский контекст (в данном случае – глобальный). Этот контекст находится в области видимости multiply. Там и найдена переменная c.

```
1  let a = 5;
2  let c = 3;
3  function multiply(x) {
4    let result = x * c; /* мы здесь */
5    return result;
6  }
7  let b = multiply(a);
8  console.log("выводим результат: " + b);
```



Как выполняется код: пример 2

Дальше всё происходит точно как в первом примере: подсчитывается и возвращается result, контекст multiply очищается и уходит с вершины стека, b получает новое значение, а потом это значение складывается со строкой и выводится в консоль.

```
let a = 5;
let c = 3;
function multiply(x) {
  let result = x * c;
  return result;
}
let b = multiply(a);
console.log("выводим результат: " + b);
// "выводим результат: 15"
```

Как выполняется код: пример 3

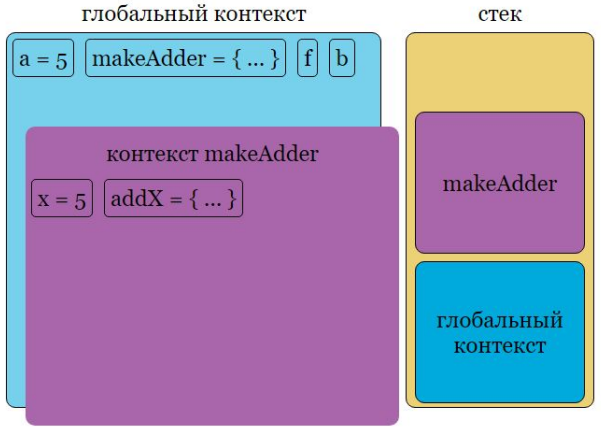
Ещё раз усложним исходный код. На этот раз создадим функцию высшего порядка, которая создаёт новую функцию и возвращает её.

```
1  let a = 5;
2  function makeAdder(x) {
3      function addX(y) {
4          return x + y;
5      }
6      return addX;
7  }
8  let f = makeAdder(a);
9  let b = f(3);
10 console.log("выводим результат: " + b);
```

Как выполняется код: строка 5

Начало вновь проходит аналогично. После того как вызывается функция makeAdder, в её лексическом окружении появляются аргумент x и вложенная функция addX. Именно она и будет возвращена в качестве значения для присвоения в f.

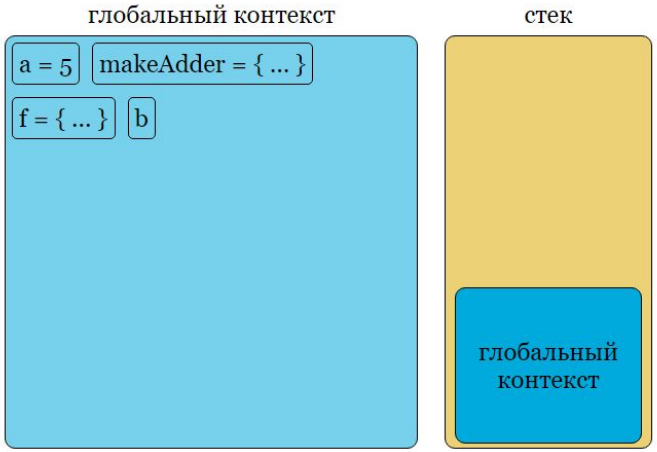
```
1  let a = 5;
2  function makeAdder(x) {
3      function addX(y) {
4          return x + y;
5      } /* мы здесь */
6      return addX;
7  }
8  let f = makeAdder(a);
9  let b = f(3);
10 console.log("выводим результат: " + b);
```



Как выполняется код: строка 9

Наконец, на строке 9 интерпретатор вызывает функцию f с аргументом 3. Контекст makeAdder к этому моменту должен быть очищен.

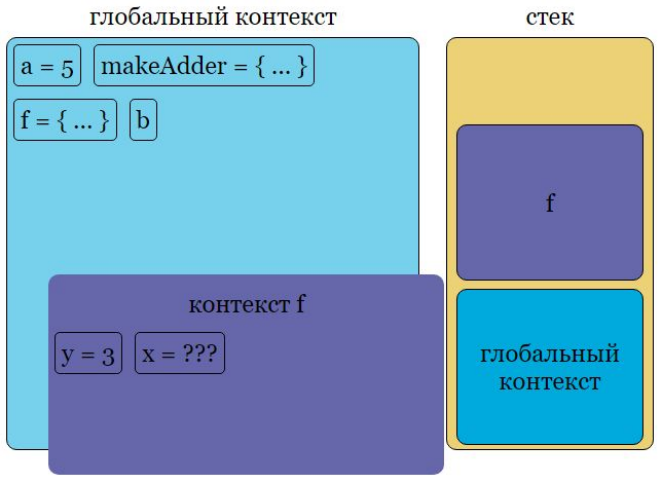
```
1  let a = 5;
2  function makeAdder(x) {
3      function addX(y) {
4          return x + y;
5      }
6      return addX;
7  }
8  let f = makeAdder(a);
9  let b = f(3); /* мы здесь */
10 console.log("выводим результат: " + b);
```



Как выполняется код: строки 3-4

Функция f вызывается, её контекст создаётся и помещается на вершину стека. В её лексическом окружении появляется аргумент y. Однако на строке 4 интерпретатор видит обращение к переменной x, которой в контексте нет.

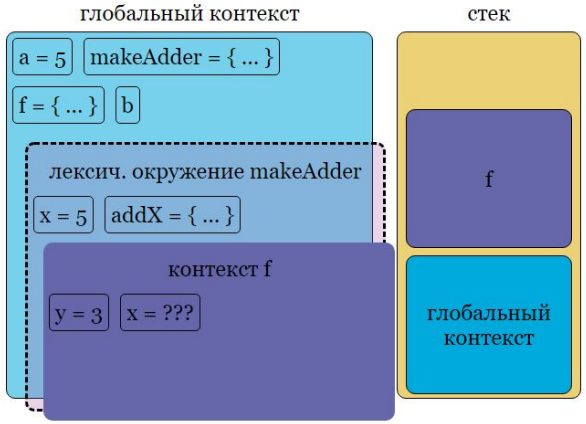
```
1  let a = 5;
2  function makeAdder(x) {
3      function addX(y) {
4          return x + y; /* мы здесь */
5      }
6      return addX;
7  }
8  let f = makeAdder(a);
9  let b = f(3);
10 console.log("выводим результат: " + b);
```



Как выполняется код: строка 4, часть 2

Тут происходит главный трюк. Оказывается, функция f заблаговременно сохранила ссылку на лексическое окружение родителя, makeAdder, а потому может поискать переменную x там. Таким образом, контекст makeAdder не был полностью удалён!

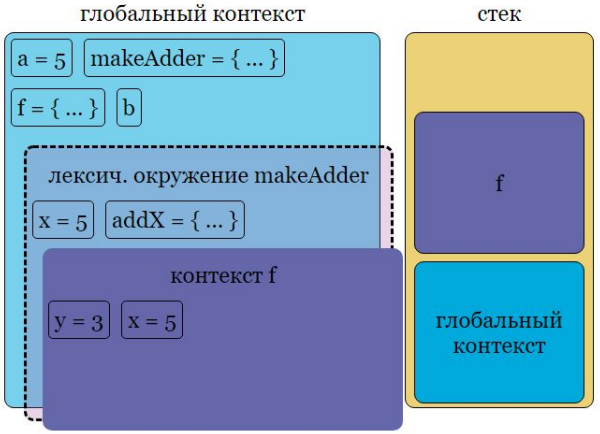
```
1  let a = 5;
2  function makeAdder(x) {
3      function addX(y) {
4          return x + y; /* мы здесь */
5      }
6      return addX;
7  }
8  let f = makeAdder(a);
9  let b = f(3);
10 console.log("выводим результат: " + b);
```



Как выполняется код: строка 4, часть 3

Дальше программа может выполняться нормально: значения всех нужных переменных доступны. Всё это – за счёт того, что функция f ещё при создании «запомнила» окружение, в котором она создавалась.

```
1  let a = 5;
2  function makeAdder(x) {
3      function addX(y) {
4          return x + y; /* мы здесь */
5      }
6      return addX;
7  }
8  let f = makeAdder(a);
9  let b = f(3);
10 console.log("выводим результат: " + b);
```



Замыкания / closures

Замыкание – это комбинация функции и лексического окружения, в котором она была создана.

Ещё говорят, что замыкание – это функция, которая ссылается **на свободные переменные**. Свободные – те, которые не были переданы в неё как параметры и не были объявлены непосредственно в теле функции.

Другими словами, функция, определённая в замыкании, «запоминает» окружение, в котором она была создана.

Замыкания: пример 1

```
1  function generateCounter() {
2    var counter = 0;
3    const addOne = function() {
4      counter++;
5      return counter;
6    }
7    return addOne;
8  }
9
10 var increment = generateCounter();
11 console.log(increment()); // 1
12 console.log(increment()); // 2
13 var increment2 = generateCounter();
14 console.log(increment2()); // 1, независимо!
```

Замыкания: пример 2

```
1  function powerFactory(exp) {
2      function op(x) {
3          return x ** exp;
4      }
5      return op;
6  }
7
8  let square = powerFactory(2);
9  let cube = powerFactory(3);
10 console.log(square(5)); // 25
11 console.log(cube(3)); // 27
12 console.log(cube(5)); // 125
```

Анонимные функции

Функции, которые передаются в аргументы или возвращаются из других функций, можно не записывать в переменную, а создавать «на лету». Такие функции называют анонимными (поскольку они не привязаны к «имени» –идентификатору/переменной).

```
1  function whatToDoWithX(x, callback) {
2      return callback(x);
3  }
4
5  console.log(whatToDoWithX(26, function(n) {
6      return Math.floor(Math.sqrt(n));
7  })); // 5
```

Методы массивов с обратным вызовом

У массивов есть несколько удобных методов, которые в качестве параметров принимают другие функции:

- forEach
- filter
- map
- reduce
- every / some
- find / findIndex

forEach

Метод forEach(callback) перебирает массив и для каждого элемента вызывает функцию callback.

Эта функция получает три параметра callback(item, index, arr):

- item – текущий элемент массива
- index – его индекс
- arr – исходный массив, который мы перебираем.

forEach: пример 1

```
1  let arr = [1, 2, 3, 4, 5];
2
3  // используем анонимную функцию
4  arr.forEach(function(elem, index, arr) {
5      console.log("element " + elem + " at index " + index);
6  });
7  // element 1 at index 0
8  // element 2 at index 1
9  // element 3 at index 2
10 // element 4 at index 3
11 // element 5 at index 4
12
13 // в callback можно указывать не все параметры, а только нужные
14 arr.forEach(function(elem) {
15     console.log(elem * elem);
16 })
17 // 1 4 9 16 25
```

forEach: пример 2

```
let numbers = [5, 4, 3, 2];
function multiplyArrayElem(x) {
    const multi = function(elem, index, arr) {
        arr[index] = elem * x;
    }
    return multi;
}

let x2 = multiplyArrayElem(2); // получаем функцию
numbers.forEach(x2);           // и передаём
console.log(numbers); // [10, 8, 6, 4]

numbers.forEach(multiplyArrayElem(0.5)); // сразу получаем и передаём
console.log(numbers); // [5, 4, 3, 2]
```

Array.filter

Метод filter(callback) используется для **фильтрации** массива. Он перебирает исходный массив, вызывает функцию callback для каждого элемента и возвращает новый массив, состоящий из **некоторых** элементов исходного. Если callback для элемента возвращает true, этот элемент включается в новый массив, если false – нет.

Параметры callback те же, что для forEach:

- item – текущий элемент массива
- index – его индекс
- arr – исходный массив, который мы перебираем.

filter: пример 1

```
1  let nums = [0, -1, 2, -4, 4, 3];
2
3  let positiveFilter = function(num) {
4      return (num > 0 ? true : false);
5  };
6
7  let oddFilter = function(num) {
8      return (num % 2 ? true : false);
9  }
10
11 let posNums = nums.filter(positiveFilter);
12 console.log(posNums); // [2, 4, 3] - только положительные
13 let oddNums = nums.filter(oddFilter);
14 console.log(oddNums); // [-1, 3] - только нечётные
```

filter: пример 2

```
let cities = [
    {name: "Yaoundé", country: "Cameroon", population: 3.0},
    {name: "Cairo", country: "Egypt", population: 17.1},
    {name: "Abidjan", country: "Ivory Coast", population: 5.2},
    {name: "Alexandria", country: "Egypt", population: 5.3},
    {name: "Dakar", country: "Senegal", population: 3.4}
];

let lessThan5Million = cities.filter(function(elem) {
    return elem.population < 5;
});
lessThan5Million.forEach(function(elem) { console.log(elem.name) });
// Yaoundé  Dakar
let egyptOnly = cities.filter(function(elem) {
    return elem.country == "Egypt";
});
egyptOnly.forEach(function(elem) { console.log(elem.name) });
// Cairo  Alexandria
```

Array.map

Метод map(callback) **трансформирует** массив. Он перебирает все элементы исходного массива, вызывает функцию callback для каждого элемента и возвращает новый массив, состоящий из **результатов выполнения callback**.

Параметры callback всё те же:

- item – текущий элемент массива
- index – его индекс
- arr – исходный массив, который мы перебираем.

map: пример 1

```
1  let strings = ["Один", "дВА", "три", "ЧЕТЫРЕ"];
2
3  let lowerStrings = strings.map(function(s) {
4      return s.toLowerCase();
5  });
6  console.log(lowerStrings); // ["один", "два", "три", "четыре"]
7
8  let firstLetters = lowerStrings.map(function(elem) {
9      return elem[0];
10 });
11 console.log(firstLetters); // ["о", "д", "т", "ч"]
```

map: пример 2

```
let revenues = [
    "Activision Blizzard: 6513", "EA: 5095",
    "Tencent: 18120", "Nintendo: 3158"
];

let revObjects = revenues.map(function(elem) {
    let revObj = {};
    let splitter = elem.indexOf(": ");
    if (splitter != -1) {
        revObj.name = elem.slice(0, splitter);
        revObj.revenue2017 = elem.slice(splitter+2);
    }
    return revObj;
});
console.log(revObjects);
// [ {name: "Activision Blizzard", revenue2017: "6513"},
//   {name: "EA", revenue2017: "5095"},
//   {name: "Tencent", revenue2017: "18120"},
//   {name: "Nintendo", revenue2017: "3158"} ]
```

Array.reduce

Метод reduce(callback) делает **свёртку массива** – подсчёт одного значения на основе перебора всех элементов. Он перебирает все элементы исходного массива, вызывает функцию callback для каждого элемента, а возвращает результат свёртки (т.е. одно итоговое значение).

Параметры callback(previousValue, item, index, arr):

- previousValue – промежуточный результат свёртки
- item – текущий элемент массива
- index – его индекс
- arr – исходный массив, который мы перебираем.

reduce: пример 1

```
1  let randArr = [];
2  for (let i = 0; i < 10; i++) {
3      randArr[i] = Math.ceil(Math.random() * 20);
4  }
5  // заполнили массив 10 случайными числами от 1 до 20
6  // например, [9, 9, 5, 2, 12, 3, 14, 15, 17, 4]
7  let sum = randArr.reduce(function(currSum, elem) {
8      return currSum + elem;
9  });
10 console.log(sum); // 90
11 let biggest = randArr.reduce(function(big, elem) {
12     return (elem > big ? elem : big);
13 });
14 console.log(biggest); // 17
```

Необязательный аргумент reduce

reduce может вызываться с двумя аргументами:

```
reduce(callback, initialValue)
```

Если второй аргумент указан, то при первом вызове callback previousValue получает значение initialValue. Если он не указан, то в previousValue записывается первый элемент массива, а перебор начинается со второго элемента.

reduce: пример 2

```
1  let letters = ["a", "c", "c", "d", "a", "e", "f", "e", "b", "d"];
2
3  function addOnlyUnique(arr, elem) {
4      if (arr.indexOf(elem) === -1)
5          arr.push(elem);
6      return arr;
7  }
8
9  let uniqueLetters = letters.reduce(addOnlyUnique, []);
10         // второй аргумент: начальное значение - пустой массив
11 console.log(uniqueLetters); // ["a", "c", "d", "e", "f", "b"]
```

Array.reduceRight

Метод reduceRight(callback[, initialValue]) делает точно то же, что reduce, но движется по массиву от конца к началу.

```
let letters = ["a", "c", "c", "d", "a", "e", "f", "e", "b", "d"];

function addOnlyUnique(arr, elem) {
    if (arr.indexOf(elem) === -1)
        arr.push(elem);
    return arr;
}

let uniqueLetters = letters.reduceRight(addOnlyUnique, []);
console.log(uniqueLetters); // ["d", "b", "e", "f", "a", "c"]
```

Array.every / Array.some

Методы every(callback) и some(callback) используются для **проверки** массива. Оба метода вызывают callback для **каждого** элемента массива.

every возвращает true, если callback возвращает true из каждого вызова.
some возвращает true, если callback возвращает true **хотя бы для одного** вызова.
Параметры callback – те же, что у forEach:

- item – текущий элемент массива
- index – его индекс
- arr – исходный массив, который мы перебираем.

every / some: примеры

```
1  let zerosOnes = [0, 1, 1, 0, 1, 1, 1, 0, 1];
2  let almostGood = [1, 0, 1, 0, 2, 1, 1, 0, 1];
3
4  function isBinaryNumber(num) {
5      return ((num == 0)||(num == 1));
6  }
7  console.log(zerosOnes.every(isBinaryNumber)); // true
8  console.log(almostGood.every(isBinaryNumber)); // false
9
10 let dump = ["some string", false, {key: "value"}];
11
12 function searchForObject(val) {
13     return (typeof val == typeof {});
14 }
15 console.log(dump.some(searchForObject));      // true
16 console.log(zerosOnes.some(searchForObject)); // false
```

Array.find / Array.findIndex

Кроме indexOf и includes, для поиска в массиве существуют методы find(callback) и findIndex(callback).

find возвращает **значение** первого элемента массива, для которого callback вернул true (а если такой элемент не найден — undefined). findIndex возвращает **индекс** первого «подходящего» элемента (или -1, если элемент не найден).

Параметры callback – вновь те же, что у forEach:

- item – текущий элемент массива
- index – его индекс
- arr – исходный массив, который мы перебираем.

find / findIndex: примеры

```
let squad = [
    { name: 'Sergio Agüero', age: 32, club: 'Man City' },
    { name: 'Exequiel Palacios', age: 21, club: 'Bayer Leverkusen' },
    { name: 'Leandro Paredes', age: 26, club: 'PSG' },
    { name: 'Lionel Messi', age: 33, club: 'Barcelona' }
];

squad.find(function(player) {
    return player.age > 30;
}); // { name: 'Sergio Agüero', age: 32 }

squad.findIndex(function(player) {
    return player.club == 'Juventus';
}); // -1

squad.findIndex(function(player) {
    return player.club == 'Man City';
}); // 0
```

Array.sort(comparator)

Метод sort тоже может вызываться с функцией обратного вызова в качестве аргумента.
Функция comparator(elem1, elem2) определяет, как сравнить любые два элемента массива:

- если функция возвращает число меньше 0, то elem1 < elem2
- если функция возвращает число больше 0, то elem1 > elem2
- если функция возвращает 0, то elem1 == elem2

sort попарно сравнивает все элементы массива, используя comparator, и переставляет их так, чтобы массив получился отсортированным.

sort(comparator): пример

```
1  let randArr = [];
2  for (let i = 0; i < 10; i++) {
3      randArr[i] = Math.ceil(Math.random() * 20);
4  } // например, [8, 13, 17, 4, 19, 6, 17, 11, 4, 3]
5
6  console.log(randArr.sort()); // попытка сортировать
7  // [11, 13, 17, 17, 19, 3, 4, 4, 6, 8] - сравниваются как строки, плохо :(
8
9  function asNumbers(first, second) {
10     if (first > second) return 1;
11     else if (first < second) return -1;
12     else return 0;
13 }
14 console.log(randArr.sort(asNumbers)); // [3, 4, 4, 6, 8, 11, 13, 17, 17, 19]
15 // а можно ещё проще!
16 console.log(randArr.sort(function(a,b) { return a-b; }));
```

Практика

1. Напишите функцию counterFactory(start, step), которая при вызове возвращает другую функцию – счётчик tictoc(). start – стартовое значение счётчика, step – его шаг. При каждом вызове tictoc увеличивает значение счётчика на step.
2. Напишите функцию take(tictoc, x), которая вызывает функцию tictoc заданное число (x) раз и возвращает массив с результатами вызовов.

Практика

3. Разбейте текст этой задачи на отдельные слова, удаляя по пути точки и запятые, а полученные слова сложите в массив. Напишите функцию, которая возвращает массив из тех же слов, но развёрнутых задом наперёд, причём массив должен быть отсортирован по количеству букв в слове. Напишите другую функцию, которая считает общее количество букв «с» во всех элементах массива.

Внеклассное чтение

https://proglib.io/p/js-closures-1/ и https://proglib.io/p/js-closures-2/
https://learn.javascript.ru/closure
https://habr.com/ru/company/plarium/blog/428612/
https://learn.javascript.ru/array-methods