



Факультет компьютерных наук

Департамент программной
инженерии

Москва
2025

Итераторы.
Коллекции.
Регулярные выражения.



Итераторы

Итератор – объект, который предоставляет метод `next()`, возвращающий следующий элемент последовательности, и флаг, характеризующий завершение итерации.

После создания, объект-итератор может быть явно использован, с помощью вызовов метода `next()`.

Любой итератор одноразовый, т. е. нельзя использовать один и тот же объект для многократной сбрасываемой итерации.

Когда итератор перебрал все значения, его поле `value` при следующем вызове `next()` должно быть `undefined`.

```
1  function makeIterator(array) {
2      var nextIndex = 0;
3
4      return {
5          next: function () {
6              return nextIndex < array.length
7                  ? { value: array[nextIndex++], done: false }
8                  : { done: true };
9          },
10     };
11 }
12
13 var it = makeIterator(["yo", "ya"]);
14 console.log(it.next().value); // 'yo'
15 console.log(it.next().value); // 'ya'
16 console.log(it.next().done); // true
17
```



Генераторы

Генератор – это функция, возвращающая объект-итератор, которая сохраняет свой контекст между вызовами.

При вызове такой функции будет создан объект-итератор, который в свою очередь при вызове `next()` будет исполняться до тех пор, пока не встретит ключевое слово `yield`, обозначающее возврат значения. Если возвращаемым значением должно быть что-то из другого генератора, то используется `yield* otherGen()`, где `otherGen()` это другой генератор.

```
1  function* idMaker() {  
2      var index = 0;  
3      while (index < 3) yield index++;  
4  }  
5  
6  var gen = idMaker();  
7  
8  console.log(gen.next().value); // 0  
9  console.log(gen.next().value); // 1  
10 console.log(gen.next().value); // 2  
11 console.log(gen.next().value); // undefined
```



```
1  function* anotherGenerator(i) {  
2      yield i + 1;  
3      yield i + 2;  
4      yield i + 3;  
5  }  
6  
7  function* generator(i) {  
8      yield i;  
9      yield* anotherGenerator(i);  
10     yield i + 10;  
11  }  
12  
13  var gen = generator(10);  
14  
15  console.log(gen.next().value); // 10  
16  console.log(gen.next().value); // 11  
17  console.log(gen.next().value); // 12  
18  console.log(gen.next().value); // 13  
19  console.log(gen.next().value); // 20  
20
```



```
1  function* generatorFunction() {  
2      let x = yield "Введите первое число";  
3      let y = yield "Введите второе число";  
4      yield `Сумма: ${x + y}`;  
5  }  
6  
7  const generator = generatorFunction();  
8  
9  console.log(generator.next().value); // "Введите первое число"  
10 console.log(generator.next(5).value); // "Введите второе число"  
11 console.log(generator.next(10).value); // "Сумма: 15"  
12 console.log(generator.next().done); // true  
13
```

Итерируемые объекты

Объект является итерируемым, если в нем реализован механизм перебора собственных значений.

Чтобы быть итерируемым, объект обязан реализовать метод `@@iterator`. Это означает, что либо сам объект, либо какой-либо объект в его цепочке прототипов, обязан иметь свойство с именем `Symbol.iterator`, которое является итератором.

```
1  var myIterable = {};  
2  
3  myIterable[Symbol.iterator] = function* () {  
4      yield 1;  
5      yield 2;  
6      yield 3;  
7  };  
8  
9  [...myIterable]; // [1, 2, 3]  
10
```



Задание на семинар номер 1

Реализовать итерируемый объект, итератор которого умеет считать числа Фибоначчи при помощи суммы.

- Каждый n -ный вызов `next()` должен возвращать n -ое число Фибоначчи
- Если в `next()` передать `true`, мы сбрасываем итератор в начальное значение
- Генератор умеет считать только первые 10 чисел

Если будет сложно, будем писать это вместе.



```
1 function* fibonacci() {
2     let fn1 = 1;
3     let fn2 = 1;
4     let i = 0;
5     while (true) {
6         let current = fn2;
7         fn2 = fn1;
8         fn1 = fn1 + current;
9         let reset = yield current;
10        if (reset) {
11            fn1 = 1;
12            fn2 = 1;
13            i = 0;
14        }
15        i++;
16
17        if (i === 10) {
18            return
19        }
20    }
21 }
22 }
```

```
1 const obj = {
2     [Symbol.iterator]: fibonacci
3 }
4
5 for (let value of obj) {
6     console.log(value);
7 }
```




Встроенные коллекции в JS

Помимо простых объектов и массивов, JavaScript предоставляет доступ к следующим структурам данных:

- Map
- Set
- WeakMap
- WeakSet

EXPLORING JAVASCRIPT
SET() AND **MAP()**

Словарь

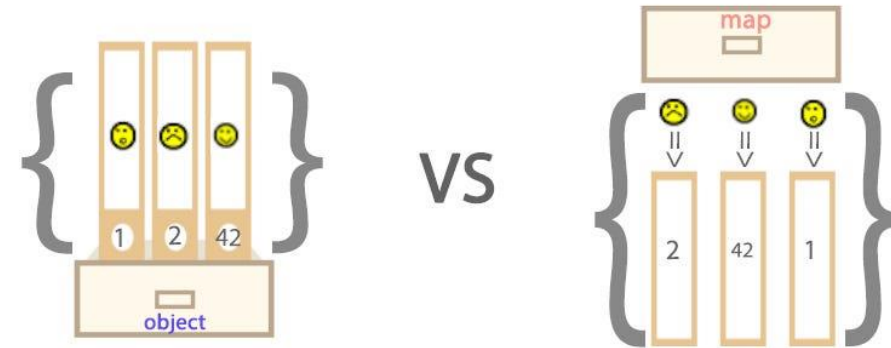
Мар – реализация простого ассоциативного массива, который содержит в себе пары ключ-значение, подобно простому объекту в JS. Помимо самих пар ключ-значение, такой объект также предоставляет доступ к методам для манипулирования данными внутри себя.

Мар похож на простой объект тем, что:

- Выполнять CRUD данных по ключу
- Проверять наличие ключа
- Итерироваться по ключам

Мар имеет следующие преимущества перед простым объектом:

- Ключами умеют быть не только строки, числа и бинарные значения
- Мар знает свою мощность своего множества пар ключ-значение
- Итерацию по данным словарь выполняет в порядке хронологического добавления данных начиная с самых старых
- Словарь не использует свои прототипные данные как внутренние данные



Как создавать словарь

Поскольку Map это внутренний функционал самого JS, который является объектом, его следует вызывать через конструктор. (На самом деле Map это класс, который разработчики JS написали сами).

Что мы можем передать в конструктор при создании словаря:

- Ничего, тогда создастся пустой словарь
- Что-то, по чему можно итерироваться и получать пары ключ-значение (например двумерный массив 2×N)
- Как вариант можно передать в аргумент объект, но поскольку простые объекты не умеют итерироваться сами по себе, то их нужно превратить во что-то итерируемое, например при помощи метода `Object.entries()`

Если в конструктор передать что-то неитерируемое, то будет выброшена ошибка `TypeError (object is not iterable)`.

```

1  const myOwnMap = new Map();
2
3  const anotherOne = new Map([
4      ['key1', 'value1'],
5      ['key2', 'value2']
6  ]);
7
8  const obj = {
9      prop: 42,
10 };
11
12 const mapByGivenObject = new Map(
13     Object.entries(obj)
14 );
  
```



Как итерироваться по словарю

По словарю можно итерироваться при помощи:

- Цикла `for ... of` – в переменную цикла будет помещаться кортеж ключ-значение
- Методом `forEach` (*спойлер*)
- Методом `entries` (*еще один спойлер*)



```
1  const anotherOne = new Map([
2    ['key1', 'value1'],
3    ['key2', 'value2']
4  ]);
5
6  for (const a of anotherOne) {
7    console.log(a);
8
9    // Сначала тут будет ['key1', 'value1']
10   // Затем ['key2', 'value2']
11  }
```



Методы Map

Словарь обладает следующими методами для манипуляции и итерации данных:

- `clear()`
- `delete()`
- `entries()`
- `forEach()`
- `get()`
- `has()`
- `keys()`
- `set()`
- `values()`

И одним свойством:

- `size`



```
1  const anotherOne = new Map([
2    ['key1', 'value1'],
3    ['key2', 'value2']
4  ]);
5
6  for (const a of anotherOne) {
7    console.log(a);
8
9    // Сначала тут будет ['key1', 'value1']
10   // Затем ['key2', 'value2']
11  }
```



get() и has()

Метод `get()` объекта-словаря проверяет наличие заданного ключа и возвращает по нему значение. Если значение по ключу не найдено, метод возвращает `undefined`.

Метод `has()` объекта-словаря проверяет наличие заданного ключа в словаре и возвращает `boolean`, отражающий его наличие.

Поскольку в словаре ключом может быть в том числе и объект, Словарь при сравнении значений использует стратегию равенства одинаковых величин (в основе которого лежат манипуляции со строгим и нестрогим равенством). Поэтому если в качестве ключа необходимо использовать объект, то придется подумать о том, как не потерять ссылку на него. Иными словами, надо помнить, что сравнение двух идентичных объектов через `===` вернет `false`, потому что это разные ячейки памяти.

```
1  const myOwnMap = new Map();
2
3  const anotherOne = new Map([
4    ['key1', 'value1'],
5    ['key2', 'value2']
6  ]);
7
8  const valueByKey = anotherOne.get('key1');
9  // value1
10
11 const unknownKey = anotherOne.get('key');
12 // undefined
13
14 const hasKey = anotherOne.has('key');
15 // false
```



set()

Метод `set()` объекта-словаря устанавливает по заданному ключу указанное значение. Если такой ключ уже есть в словаре, то он будет перезаписан. Возвращает ссылку на измененный словарь.

Благодаря возвращаемому значению, метод `set()` можно *чейнить*.

```
1  const myOwnMap = new Map([
2    ['key1', 'value1'],
3    ['key2', 'value2'],
4    ['key3', 'value3']
5  ]);
6
7  myOwnMap
8    .set('key4', 'value4')
9    .set('key5', 'value5');
```

```
1  const myOwnMap = new Map();
2
3  const anotherOne = new Map([
4    ['key1', 'value1'],
5    ['key2', 'value2']
6  ]);
7
8  const ourKey = {param: 'wftValue'}
9
10 anotherOne.set(ourKey, 'kekW');
11
12 console.log(anotherOne.get(ourKey));
13 // 'kekW'
```



keys(), values() и entries()

Метод `keys()` объекта-словаря возвращает *итератор*, который содержит все существующие на момент вызова ключи словаря.

Метод `values()` объекта-словаря возвращает *итератор*, который содержит все существующие на момент вызова значения словаря.

Метод `entries()` объекта-словаря возвращает *итератор*, который содержит все существующие на момент вызова пары ключ-значение словаря в виде массива.

```
1  const myOwnMap = new Map([
2    ['key1', 'value1'],
3    ['key2', 'value2'],
4    ['key3', 'value3']
5  ]);
6
7  for (let key of myOwnMap.keys()) {
8    console.log(key);
9  }
```




forEach()

Метод `forEach()` объекта-словаря выполняет переданную в аргументы функцию для каждой пары ключ-значение.

Передаваемая функция содержит до трех аргументов:

- Первый для значения из пары
- Второй для ключа из пары
- Третий для ссылки на словарь

Благодаря третьему аргументу словарь можно мутировать при помощи всех доступных словарю методов.

Ключи также перебираются в хронологическом порядке, начиная с самых старых.

```
1 function doMutations(value, key, map) {  
2     if (typeof value === 'number') {  
3         map.set(key, value + 1);  
4     }  
5 }  
6  
7 const m = new Map([  
8     ['foo', 3],  
9     ['bar', {}],  
10    ['baz', undefined],  
11  ]);  
12  
13 m.forEach(doMutations);  
14  
15 console.log([ ...m.values()])  
16 // [4, {}, undefined]
```



delete() и clear()

Метод `delete()` объекта словаря удаляет пару ключ-значение по заданному ключу. Если нашелся такой ключ, то всегда происходит удаление. Возвращает `boolean`, говорящий о том, была ли пара ключ-значение удалена.

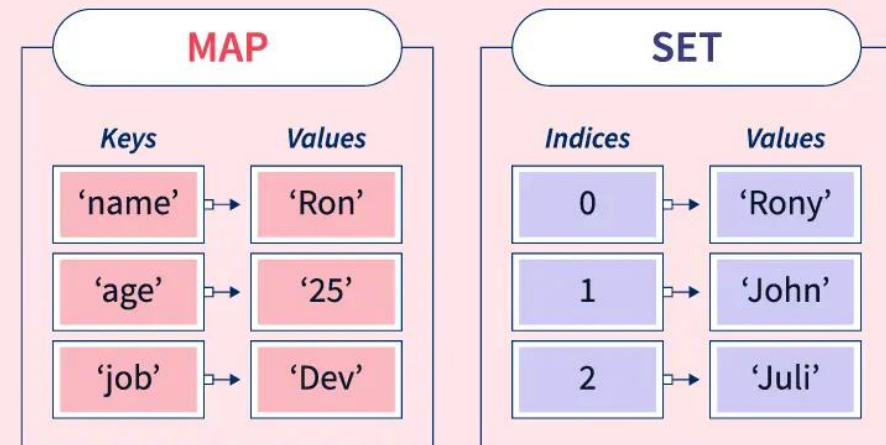
Метод `clear()` уничтожает все пары ключ-значение словаря.

```
1  const myOwnMap = new Map([
2    ['key1', 'value1'],
3    ['key2', 'value2'],
4    ['key3', 'value3']
5  ]);
6
7  console.log(myOwnMap.delete('key1'));
8  // true
9  console.log(myOwnMap.has('key1'));
10 // false
11
12 myOwnMap.clear();
13 console.log(myOwnMap);
14 // Map(0) {}
15
```

Множество

Множество – коллекция значений в виде объекта, каждое значение из которых существует в коллекции в единственном экземпляре.

Множество является итерируемым объектом, порядок значений во время перебора определяется хронологией их добавления. Так же, как и в словарях, идентичные по значению объекты не являются одинаковыми для JS в виду специфики хранения объектов в памяти.





Методы множеств

В базовом наборе множеств (до 2024 года) существовало 7 методов и 1 свойство:

- `add()`
- `clear()`
- `delete()`
- `entries()`
- `forEach()`
- `has()`
- `values()` / `keys()`
- `size`

Method	Return type	Mathematical equivalent	Venn diagram
<code>A.difference(B)</code>	Set	$A \setminus B$	
<code>A.intersection(B)</code>	Set	$A \cap B$	
<code>A.symmetricDifference(B)</code>	Set	$(A \setminus B) \cup (B \setminus A)$	
<code>A.union(B)</code>	Set	$A \cup B$	
<code>A.isDisjointFrom(B)</code>	Boolean	$A \cap B = \emptyset$	
<code>A.isSubsetOf(B)</code>	Boolean	$A \subseteq B$	
<code>A.isSupersetOf(B)</code>	Boolean	$A \supseteq B$	



Конструктор множества

Как и в словарях, для создания множества нужно использовать его конструктор. Аргумент необязателен, но если он есть, он должен быть итерируемым, иначе мы получим `TypeError`.



```
1  const set1 = new Set();  
2  const set2 = new Set([13, 42, 228, 322]);  
3  // const set3 = new Set({});  
4  // bruh  
5
```



add() и clear()

Метод `add()` объекта-множества добавляет новое значение в конец множества. Этот метод тоже *чейнится*.



```
1  const set1 = new Set();
2
3  set1.add(42);
4  set1.add(42)
5      .add(13);
6
7  for (const item of set1) {
8      console.log(item);
9      // Expected output: 42
10     // Expected output: 13
11 }
```



add(), clear() и delete()

Метод `add()` объекта-множества добавляет новое значение в конец множества. Этот метод тоже *чейнится*.

Метод `clear()` очищает множество.

Метод `delete()` удаляет элемент из множества, если он существует. Возвращает `true` если элемент был до удаления, `false` в противном случае.

```
1  const set1 = new Set();
2
3  set1.add(42);
4  set1.add(42)
5      .add(13);
6
7  for (const item of set1) {
8      console.log(item);
9      // Expected output: 42
10     // Expected output: 13
11 }
```



values(), keys() и entries()

Методы `values()` и `keys()` возвращают итератор, который позволяет перебрать все существующие во множестве значения в порядке их попадания во множество.

Метод `entries()` возвращает итератор, результат `next()` которого возвращает пару значение-значение.

```
1  const set1 = new Set();
2  set1.add(42);
3  set1.add('forty two');
4
5  const iterator1 = set1.entries();
6
7  for (const entry of iterator1) {
8    console.log(entry);
9    // Expected output: Array [42, 42]
10   // Expected output: Array ["forty two", "forty two"]
11  }
12
```




forEach()

Метод `forEach()` позволяет выполнить передаваемую в аргумент функцию для всех значений множества. По аналогии со словарем, этим методом также можно мутировать множество в процессе перебора значений для выполнения передаваемой функции.

```
1 function logSetElements(value1, value2, set) {  
2     console.log(`s[${value1}] = ${value2}`);  
3 }  
4  
5 new Set(['foo', 'bar', undefined]).forEach(logSetElements);  
6  
7 // Expected output: "s[foo] = foo"  
8 // Expected output: "s[bar] = bar"  
9 // Expected output: "s[undefined] = undefined"  
10
```



has()

Метод `has()` возвращает `boolean` в зависимости от того, был ли найден передаваемый элемент во множестве.

```
1  const set1 = new Set([1, 2, 3, 4, 5]);
2
3  console.log(set1.has(1));
4  // Expected output: true
5
6  console.log(set1.has(5));
7  // Expected output: true
8
9  console.log(set1.has(6));
10 // Expected output: false
11
```



WeakMap

WeakMap – это объект-словарь, ключами которого могут быть только объекты и незарезервированные Symbol со значениями произвольных типов.

Основное отличие WeakMap от обычных словарей заключается в том, что при удалении ключа сборщиком мусора, он также пропадет и из WeakMap вместе с привязанным к нему значением. Это бывает удобно, когда мы храним какую-либо информацию, имеющую отношение к какому-либо объекту, и при его удалении сборщиком мусора нам не имеет смысла держать данные, зависящие от удаленного объекта.

У WeakMap отсутствует итератор, поскольку в случае его существования результат работы итератора зависел бы от состояния сборщика мусора и порождал бы неконтролируемое поведение кода. Поэтому у WeakMap есть только методы `get()`, `set()`, `has()` и `delete()`.



```
1  const privates = new WeakMap();
2
3  export class Public {
4    constructor() {
5      const me = {
6        // Приватные данные идут здесь
7      };
8      // 'me' будет освобождён вместе с 'this' !!!
9      privates.set(this, me);
10   };
11
12   method() {
13     const me = privates.get(this);
14     // Сделайте что-нибудь с приватными данными в 'me' ...
15   };
16 }
17
```



WeakSet

WeakSet – это объект-множество, значениями которого могут быть только объекты и незарезервированные Symbol.

WeakSet ведет себя подобно WeakMap: так же не умеет итерироваться, можно пользоваться только методами add(), delete() и has().

Это множество полезно в случаях, когда нам необходимо отслеживать ссылки на объекты, но не мешать их удалению сборщиком мусора, а также WeakSet (как и WeakMap) позволяет выполнять кеширование без утечек памяти.



```
1  const cache = new WeakSet();
2
3  function process(obj) {
4      if (cache.has(obj)) {
5          console.log("Уже обработан:", obj);
6          return;
7      }
8
9      cache.add(obj);
10     console.log("Обрабатываем:", obj);
11 }
12
13 let user = { name: "Alice" };
14 process(user); // Обрабатываем
15 process(user); // Уже обработан
16
17 user = null; // Объект удалится из памяти и из WeakSet
18
```



Задание на семинар 2

Задача: Учет посетителей кафе. Вам нужно реализовать систему учета посетителей кафе в виде класса с использованием Map, Set, WeakMap и WeakSet.

Условия:

- Хранение заказов (Map)
 - У каждого посетителя есть уникальный ID (number), и мы храним информацию о его заказе в Map.
 - Если посетитель делает новый заказ, он заменяет предыдущий.
- Список постоянных клиентов (Set)
 - Храним ID посетителей, которые приходили в кафе хотя бы один раз.
- Система скидок (WeakMap)
 - Если посетитель заказывает более 3-х раз, мы добавляем его в WeakMap и даем скидку.
 - Если посетитель перестает ходить в кафе, его данные автоматически удаляются.
- Клиенты в заведении прямо сейчас (WeakSet)
 - Когда посетитель приходит в кафе, мы добавляем его объект в WeakSet.
 - Когда он уходит, удаляем из WeakSet.



```
1  class Cafe {
2      constructor() {
3          this.orders = new Map(); // ID → заказ
4          this.visitors = new Set(); // Уникальные ID посетителей
5          this.discounts = new WeakMap(); // Объект посетителя → количество посещений
6          this.currentVisitors = new WeakSet(); // Посетители, которые сейчас в кафе
7      }
8
9      enter(visitor) {
10         console.log(`${visitor.name} зашел в кафе.`);
11         this.currentVisitors.add(visitor);
12         this.visitors.add(visitor.id);
13     }
14
15     leave(visitor) {
16         console.log(`${visitor.name} ушел из кафе.`);
17         this.currentVisitors.delete(visitor);
18     }
19
20     makeOrder(visitor, order) {
21         console.log(`${visitor.name} сделал заказ: ${order}`);
22         this.orders.set(visitor.id, order);
23
24         // Учет скидок
25         if (!this.discounts.has(visitor)) {
26             this.discounts.set(visitor, 1);
27         } else {
28             this.discounts.set(visitor, this.discounts.get(visitor) + 1);
29         }
30
31         if (this.discounts.get(visitor) > 3) {
32             console.log(`${visitor.name} получает скидку!`);
33         }
34     }
35 }
```



```
1 // Создаем кафе
2 const cafe = new Cafe();
3
4 // Посетители
5 let alice = { id: 1, name: "Alice" };
6 let bob = { id: 2, name: "Bob" };
7
8 // Посетители заходят в кафе
9 cafe.enter(alice);
10 cafe.enter(bob);
11
12 // Заказы
13 cafe.makeOrder(alice, "Капучино");
14 cafe.makeOrder(bob, "Эспрессо");
15 cafe.makeOrder(alice, "Латте");
16 cafe.makeOrder(alice, "Чай");
17 cafe.makeOrder(alice, "Американо"); // Alice получает скидку!
18
19 // Посетители уходят
20 cafe.leave(alice);
21 cafe.leave(bob);
22
23 // Удаляем объект Bob (например, он больше не приходит в кафе)
24 bob = null;
25
26 // Через некоторое время данные о Bob автоматически удалятся из WeakMap и WeakSet
```




Регулярные выражения

Регулярные выражения – это объекты-шаблоны, которые используются для сопоставления строк с последовательностями символов.

В JavaScript регулярные выражения можно объявить двумя способами:

1. Заклучив паттерн между слешами для независимых паттернов
2. Используя класс RegExp для зависимых паттернов

Сегодня мы не будем учиться писать регулярки, наша задача узнать, что мы умеем с ними делать с точки зрения программирования



```
1  const regexp1 = /pattern/;  
2  
3  const random = (Math.random() * 10).round() / 10;  
4  
5  const regexp2 = new RegExp(`pat${random}tern`);
```




Доступные методы

У объектов, представляющих паттерн для работы со строками, существуют 5 основных методов:

- `exec()`
- `test()`
- `match()`
- `search()`
- `replace()`

Начнем от простых к сложным



```
1  const regexp1 = /pattern/;  
2  
3  const random = (Math.random() * 10).round() / 10;  
4  
5  const regexp2 = new RegExp(`pat${random}tern`);
```



test()

Метод `test()` сопоставляет регулярному выражению строку и возвращает соответствующий `boolean`.

```
1  const regexp = /a+/;  
2  
3  console.log(regexp.test('bread'));  
4  // true  
5  
6  console.log(regexp.test('bruh'));  
7  // false
```



exec()

Метод `exec()` выполняет поиск сопоставления регулярного выражения в указанной строке. Возвращает массив с результатами или `null`.

```
1 // Сопоставляется с фразой «кайф, сплющь»,  
2 // за которой следует слово «вши»,  
3 // игнорируя любые символы между ними.  
4 // Запоминает слова «сплющь» и «вши».  
5 // Игнорирует регистр символов.  
6 var re = /кайф,\s(сплющь).+?(вши)/gi;  
7 var result = re.exec("Эх, чужд кайф, сплющь объём вши, грызя цент.");  
8
```



```
1 // Сопоставляется с фразой «кайф, сплющь»,  
2 // за которой следует слово «вши»,  
3 // игнорируя любые символы между ними.  
4 // Запоминает слова «сплющь» и «вши».  
5 // Игнорирует регистр символов.  
6 var re = /кайф,\s(сплющь).+?(вши)/gi;  
7 var result = re.exec("Эх, чужд кайф, сплющь объём вши, грызя цент.");  
8
```

Объект	Свойство/ Индекс	Описание	Пример
result	[0]	Все сопоставившиеся символы в строке.	кайф, сплющь объём вши
	[1], ...[n]	Сопоставившиеся подстроки в круглых скобках, если они присутствуют. Количество возможных подстрок ничем не ограничено.	[1] = сплющь [2] = вши
	index	Индекс сопоставления в строке, начинается с нуля.	9
	input	Оригинальная строка.	Эх, чужд кайф, сплющь объём вши, грызя цент.

re	lastIndex	Индекс, с которого начнётся следующая попытка сопоставления. Если отсутствует флаг "g", остаётся равным нулю.	31
	ignoreCase	Указывает, что в регулярном выражении используется флаг игнорирования регистра "i".	true
	global	Указывает, что в регулярном выражении используется флаг глобального сопоставления "g".	true
	multiline	Указывает, что в регулярном выражении используется флаг сопоставления по нескольким строкам "m".	false
	source	Текст шаблона регулярного выражения.	кайф,\s(сплющь). +?(вши)

match()

Метод `match()` возвращает получившиеся совпадения при сопоставлении строки с регулярным выражением.

Возвращает объект `Array`, содержащий результаты сопоставления, или `null`, если ничего не было сопоставлено.

```

1  var str = "Смотри главу 3.4.5.1 для дополнительной информации";
2  var re = /смотри (главу \d+(\.\d)*)/i;
3  var found = str.match(re);
4
5  console.log(found);
6
7  // выведет [ 'Смотри главу 3.4.5.1',
8  //           'главу 3.4.5.1',
9  //           '.1',
10 //           index: 0,
11 //           input: 'Смотри главу 3.4.5.1 для дополнительной информации' ]
12
13 // 'Смотри главу 3.4.5.1' - это полное сопоставление
14 // 'главу 3.4.5.1' - первое значение, сопоставленное с группой "(главу \d+(\.\d)*)".
15 // '.1' - это последнее значение, сопоставленное с группой "(\.\d)".
16 // Свойство 'index' содержит значение (0) индекса совпадения
17 // относительно начала сопоставления
18 // Свойство 'input' содержит значение введённой строки.
19

```

replace()

Метод `replace()` возвращает новую строку с некоторыми или всеми сопоставлениями с шаблоном, заменёнными на заменитель. Шаблон может быть строкой или регулярным выражением, а заменитель может быть строкой или функцией, вызываемой при каждом сопоставлении.

```

1  function replacer(match, p1, p2, p3, offset, string) {
2      // p1 - не цифры, p2 - цифры, p3 - не буквы и не цифры
3      return [p1, p2, p3].join(" - ");
4  }
5  var newString = "abc12345#$*%".replace(
6      /([^\d]*)(\d*)([^\w]*)/,
7      replacer
8  );
9
10 console.log(newString)
11 // abc - 12345 - #$*%

```



Спасибо за семинар!

Вы прекрасны!

Пройдя [эту](#) форму, вы можете получить дополнительный балл к семинару.

[Сюда](#) вы можете загрузить свои 4ые домашние задания



