



КОНТЕЙНЕРЫ



ДЕНИС ЕЖКОВ



ДЕНИС ЕЖКОВ

Frontend-разработчик в «Ростелеком IT»

[aka_SKIf](#)

facebook.com/ezhkov

[@ezhkov](#)



ПЛАН ЗАНЯТИЯ

1. [Set](#)
2. [Map](#)
3. [WeakSet, WeakMap](#)

ES6

Контейнеры



КОНТЕЙНЕР SET

ПРИМЕР

Мы делаем интернет-магазин и нам нужно реализовать фильтр по размерам. Для этого нужно определить все существующие значения поля `size` из массива объектов `products`.

```
1  const products = [  
2    {  
3      name: 'balenciaga triple s',  
4      sizes: [39, 42, 45],  
5    },  
6    ...  
7  ];
```

БЫСТРОЕ РЕШЕНИЕ

Первая мысль: положить все размеры в один массив.

```
1  const sizes = products.reduce((result, product) => {  
2    result.push(...product.sizes);  
3  
4    return result;  
5  }, []);
```

АНАЛИЗ РЕШЕНИЯ

Работает не совсем корректно, ведь в итоговом массиве будут дубли.

```
1 | console.log(sizes); // [39, 42, 45, 39, 42, 43]
```

Пробежимся по массиву `sizes` ещё раз для устранения дублей.

УБИРАЕМ ДУБЛИ

Воспользуемся свойством объекта — уникальностью ключей.

```
1  const uniqueSizes = sizes.reduce((result, size) => {  
2    result[size] = true;  
3  
4    return result;  
5  }, {});  
6  
7  console.log(Object.keys(uniqueSizes)); // ['39', '42', '43', '45'] уже строки!
```

ИТОГОВОЕ РЕШЕНИЕ

```
1  const sizes = products.reduce((result, product) => {
2    result.push(...product.sizes);
3
4    return result;
5  }, []);
6
7  const uniqueSizes = sizes.reduce((result, size) => {
8    result[size] = true;
9
10   return result;
11 }, {});
12
13 console.log(Object.keys(uniqueSizes)); // ['39', '42', '43', '45'] уже строки!
```

Примечание*: конечно же в production коде вы будете писать в одну строку.

НЕДОСТАТКИ ПОЛУЧИВШЕГОСЯ РЕШЕНИЯ

- сложный для понимания алгоритм;
- поменялся тип значения размера (стал строкой).

Конечно же, мы можем с помощью `map` пройти по массиву и переделать всё в числа.

Как исправить? Использовать нативный класс `Set`.

УПРОСТИМ СУЩЕСТВУЮЩИЙ КОД

Упростим существующий код, применив `Set`.

```
1  const sizes = products.reduce((result, product) => {  
2    product.sizes.forEach(size => result.add(size));  
3  
4    return result;  
5  }, new Set());  
6  
7  console.log([...sizes]); // // [39, 42, 43, 45] числа!
```

Размеры не поменяли тип (остались числами)!

ЧТО ЗА НОВЫЙ ТИП КОЛЛЕКЦИЙ – Set?

Это класс, позволяющий хранить только уникальные значения какого-либо типа.

```
1 | const set = new Set([1, 2, 2]);  
2 | console.log(set); // Set(2) {1, 2}
```

НО

```
1 | const set = new Set([1, 2, '2']);  
2 | console.log(set); // Set(2) {1, 2, '2'}
```

Важно, что значения одного типа.

НЮАНС РАБОТЫ С ОБЪЕКТАМИ

```
1 | const set = new Set([{foo: 'bar'}, {foo: 'bar'}]);  
2 | console.log(set.size); // 2
```

Потому что объекты в JS сравниваются по ссылкам!

```
1 | {} === {} // false  
2 | {} == {} // false
```

АЛГОРИТМ СРАВНЕНИЯ

Элементы сравниваются подобно оператору `===`, но есть исключение на `NaN`.

```
1  NaN === NaN // false
2
3  const set = new Set([NaN]);
4
5  set.size // 1
6  set.has(NaN) // true
7
8  // но при этом:
9  set.add(NaN)
10 set.size // 1
```

ИНТЕРФЕЙС КЛАССА SET

```
1  const set = new Set();
2  set.add('foo')
3  set.has('foo') // true
4  set.size // 1
5
6  set.delete('foo') // true, вернется false если изначально не было поля 'foo'
7  set.has('foo') // false
8
9  set.clear();
10 set.size // 0
```


ИТЕРАЦИЯ ПО МАССИВУ

```
1  const set = new Set([39, 42, 41]);
2
3  // 1 способ - как итератор
4  for (const size of set) {
5      console.log(size);
6  }
7
8  // 2 способ - через метод forEach
9  set.forEach(size => console.log(size))
```

ВЫВОДЫ

Зачем нужен контейнер `Set`, если у нас уже есть массивы?

👁👁 Пригодится, если необходимо выделить только уникальные значения.



КОНТЕЙНЕР МАР

ПРИМЕР

Продолжаем делать интернет-магазин. Нам нужно сохранять дополнительную информацию о пользователе.

```
1  const privateData = [];  
2  class Client {  
3    constructor(name, phone, address) {  
4      this.name = name;  
5      this.phone = phone;  
6  
7      privateData.push({  
8        name,  
9        phone,  
10       address,  
11     });  
12   }  
13   getAddress() {  
14     // ...  
15   }  
16 }
```

РЕАЛИЗАЦИЯ GETADDRESS

```
1  getAddress() {  
2    const isSuitable = data => (  
3      data.phone === this.phone &&  
4      data.name === this.name  
5    );  
6  
7    const suspected = privateData.filter(isSuitable);  
8  
9    return suspected[suspected.length - 1].address;  
10 }
```



НЕДОСТАТКИ ПОЛУЧИВШЕГОСЯ РЕШЕНИЯ

- сложный для понимания алгоритм;
- завязка на необходимое число полей (phone, name) для идентификации, при расширении кода можно потерять.

Как исправить? Использовать нативный класс `Map`.

УПРОСТИМ СУЩЕСТВУЮЩИЙ КОД

Упростим существующий код, применив `Map`.

```
1  const privateData = new Map();
2
3  class Client {
4      constructor(name, phone, address) {
5          this.name = name;
6          this.phone = phone;
7
8          privateData.set(this, { name, phone, address });
9      }
10
11     getAddress() {
12         return privateData.get(this).address;
13     }
14 }
```

ЧТО ЗА НОВЫЙ ТИП КОЛЛЕКЦИЙ – Map?

Это класс, позволяющий хранить пары ключ–значение.

Его особенности:

- любое значение может быть ключом (даже объект);
- сохраняет порядок вставки.

```
1 | const map = new Map([['foo', 1], ['bar', 2]]);  
2 | console.log(map); // Map(2) {"foo" => 1, "bar" => 2}
```


ОБЪЕКТ КАК КЛЮЧ

```
1  const map = new Map();
2  const buyerContacts = {
3    phone: 123456789,
4    email: 'test@test.ru',
5  };
6  map.set(buyerContacts, 'Ivan');
7
8  map.get(buyerContacts); // Ivan
```

НЮАНС ПРИ РАБОТЕ С ОБЪЕКТАМИ В КАЧЕСТВЕ КЛЮЧЕЙ

```
1  const map = new Map([
2    [{ foo: 1 }, 'bar'],
3    [{ foo: 1 }, 'bar'],
4  ]);
5
6  console.log(map.size); // 2
```

Опять-таки, потому что объекты в JS сравниваются по ссылкам!

```
1  {} === {} // false
2  {} == {}  // false
```

АЛГОРИТМ СРАВНЕНИЯ

Элементы сравниваются при поиске подобно оператору `===`, но есть исключение на `NaN`.

```
1 NaN === NaN; // false
2
3 const map = new Map();
4 map.set(NaN, 'test');
5
6 map.size; // 1
7 map.get(NaN); // test
```

СОХРАНЕНИЕ ПОРЯДКА ВСТАВКИ

Итерируется в том же порядке что и добавлялись ключи.

```
1  const map = new Map();
2  map.set('foo', 1);
3  map.set('bar', 2);
4
5  map.forEach((item) => {
6    console.log(item);
7  });
8  // 1
9  // 2
```

ИНТЕРФЕЙС КЛАССА Map

```
1  const map = new Map();
2  map.set('foo', 1);
3  map.set('bar', 2);
4  map.has('foo'); // true
5  map.size; // 2
6
7  [...map.keys()]; // ['foo', 'bar']
8
9  map.delete('foo'); // true
10 // вернется false если изначально не было поля 'foo'
11
12 map.has('foo'); // false
13
14 map.clear();
15 map.size; // 0
```

ВЫВОДЫ

Зачем нужен контейнер `Map`, если у нас уже есть объекты?

✂ Пригодится, если не хватает строковых ключей и необходимо знать порядок вставки ключей.



WEAK-КОНТЕЙНЕРЫ

ПРИМЕР

Вернёмся к предыдущему примеру со скрытой информацией пользователя:

```
1  const privateData = new Map();
2  class Client {
3    constructor(name, phone, address) {
4      this.name = name;
5      this.phone = phone;
6      privateData.set(this, { name, phone, address });
7    }
8    getAddress() {
9      return privateData.get(this).address;
10   }
11 }
```


ПРИМЕР

Можно заметить, что если instance `Client` будет удален, то информация по нему в `privateData` останется в памяти, так как есть ссылка из `privateData`.

Решение: использовать `WeakMap`.

WeakMap - КОНТЕЙНЕР

Тот же Map -контейнер, содержащий пары ключ–значение, но с особенностями:

- ключом могут быть только объекты;
- не препятствует сборщику мусора.

ВЗАИМОДЕЙСТВИЕ СБОРЩИКА МУСОРА И WEAKMAP - КОНТЕЙНЕРА

Из примера ранее —

```
1  const privateData = new Map();
2  class Client {
3      constructor(name, phone, address) {
4          this.name = name;
5          this.phone = phone;
6          privateData.set(this, { name, phone, address });
7      }
8      getAddress() {
9          return privateData.get(this).address;
10     }
11 }
```

ВЗАИМОДЕЙСТВИЕ СБОРЩИКА МУСОРА И WEAKMAP - КОНТЕЙНЕРА

Получаем, что при удалении экземпляра `Client`, удалится информация из `privateData`.

ОСОБЕННОСТЬ КЛЮЧЕЙ В WeakMap

Ключи `WeakMap` не перечисляемы, в силу отсутствия борьбы со сборщиком мусора (ссылки являются “слабыми”).

Если нужен список ключей, придется реализовывать отдельно.

КРАТКО 0 WeakSet

Тот же `Set` -контейнер, содержащий уникальные значения, но с особенностями:

- в качестве значений могут быть только объекты;
- не припятствует сборщику мусора.

В общем, так же, как и `Map`.

ВЫВОДЫ

Зачем нужны Weak-контейнеры?

👍 По сравнению с другими контейнерами (Set и Map), реализуют “слабые” связи, и, не препятствуя сборщику мусора, экономят память, при этом еще и упрощая код.



ИТОГИ

Сегодня мы с вами рассмотрели достаточно много важных вещей:

1. **Set**

Когда необходимо выделить только уникальные значения.

2. **Map**

Когда не хватает строковых ключей и необходимо знать порядок вставки ключей.

3. **WeakSet, WeakMap**

Когда нужно не препятствовать сборщику мусора и сэкономить память.



Задавайте вопросы и напишите отзыв о лекции!

ДЕНИС ЕЖКОВ

[aka_SKIff](#)

facebook.com/ezhkov

[@ezhkov](#)