

Цели занятия

- изучить массивы и методы для работы с ними;
- разобрать методы `Object.keys`, `Object.values`, `Object.entries`;
- познакомиться с `Map` и `Set`;

Полезные материалы

- [Подробнее про массивы в JavaScript](#)
- [Что такое стек и очередь?](#)
- [Map и Set](#)

Массивы

Объекты позволяют хранить данные со строковыми ключами. Это замечательно.

Но довольно часто нам необходима *упорядоченная коллекция* данных, в которой присутствуют 1-й, 2-й, 3-й элементы и т.д. Например, она понадобится нам для хранения списка чего-либо: пользователей, товаров, элементов HTML и т.д.

В этом случае использовать объект неудобно, так как он не предоставляет методов управления порядком элементов. Мы не можем вставить новое свойство «между» уже существующими. Объекты просто не предназначены для этих целей.

Для хранения упорядоченных коллекций существует особая структура данных, которая называется массив, `Array`.

```
let arr1 = new Array();  
let arr2 = []; // Используется чаще всего  
  
// Создали массив фруктов
```

```
let fruits = ["Яблоко", "Апельсин", "Слива"];

// Получаем элементы массива по номеру (индексу)
alert(fruits[0]); // Яблоко
alert(fruits[1]); // Апельсин
alert(fruits[2]); // Слива

// Можно заменить любой элемент
fruits[2] = "Груша"; // теперь ["Яблоко", "Апельсин", "Груша"]

// Кол-во элементов в массиве
alert(fruits.length); // 3
```

Редактирование массива

На практике массивы используются для того, чтобы хранить упорядоченный список чего-либо, например, корзина в интернет магазине это упорядоченный список товаров. И довольно часто возникает ситуация, когда мы хотим в этот список добавить новые элементы, либо удалить некоторые.

Причём довольно часто мы хотим работать либо с концом массива, либо с его началом ([подробнее про стек и очередь](#)) и поэтому для этого существуют специальные методы.

- **pop()** — удаляет последний элемент из массива и возвращает его

```
let fruits = ["Яблоко", "Апельсин", "Груша"];

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

- **push()** — добавляет элемент в конец массива

```
let fruits = ["Яблоко", "Апельсин"];

fruits.push("Груша");
```

```
alert(fruits); // Яблоко, Апельсин, Груша
```

- **shift()** — удаляет из массива первый элемент и возвращает его

```
let fruits = ["Яблоко", "Апельсин", "Груша"];

alert(fruits.shift()); // удаляем Яблоко и выводим его

alert(fruits); // Апельсин, Груша
```

- **unshift()** — добавляет элемент в начало массива

```
let fruits = ["Апельсин", "Груша"];

fruits.unshift("Яблоко");

alert(fruits); // Яблоко, Апельсин, Груша
```

Методы массивов

В реальной практике массивы используются очень часто для решения совершенно разных задач, поэтому в JavaScript существует много методов работы с массивами. Давайте разберём самые популярные:

- **map()** — принимает функцию в качестве параметра и создает новый массив с результатом вызова указанной функции для каждого элемента массива. Обратите внимание, что оригинальный массив не будет изменён.

```
let numbers = [1, 2, 3, 4];

// Возводим каждое число в квадрат
let newNumbers = numbers.map(function (n) {
    return n * n;
});
```

```
});  
  
console.log(newNumbers); // [1, 4, 9, 16]
```

Код на примере выше можно сократить и сделать более лаконичным, используя так называемые [стрелочные функции](#). Это так называемый синтаксический сахар, который не добавляет новых возможностей, поэтому их изучение выходит за пределы занятия. Тот же пример, но в сокращенном виде:

```
let numbers = [1, 2, 3, 4];  
  
// Возводим каждое число в квадрат  
let newNumbers = numbers.map((n) => n * n);  
  
console.log(newNumbers); // [1, 4, 9, 16]
```

- **forEach()** — данный метод применяет функцию к каждому элементу массива

```
let users = [  
  { id: 1, name: "Yury" },  
  { id: 2, name: "John" },  
  { id: 3, name: "Anna" },  
];  
  
// Выводим имя каждого пользователя  
users.forEach(function (user) {  
  console.log(user.name);  
});
```

- **every()** — этот метод проверяет, удовлетворяют ли ВСЕ элементы массива условию, заданному в передаваемой функции. Он вернет значение `true`, если каждый элемент совпадет с проверяемой функцией, иначе — `false`

```
let numbers = [1, 2, 5, 22, 0];  
  
// Вернёт true, если КАЖДЫЙ элемент меньше 100  
let flag = numbers.every(function (n) {
```

```
    return n < 100;
  });
console.log(flag); // true
```

- **some()** — этот метод проверяет, удовлетворяет ли какой-либо элемент массива условию, заданному в передаваемой функции. Он вернет значение true, если хотя бы один элемент совпадет с проверяемой функцией, и значение false — если нет.

```
let numbers = [3, 56, 78, 2, 102, 15];

// Вернёт true, если ХОТЯ БЫ ОДИН элемент больше 100
let flag = numbers.some(function (n) {
  return n > 100;
});

console.log(flag); // true
```

- **includes()** — возвращает значение true, если массив содержит определенный элемент, и значение false — если нет.

```
let fruits = ["Яблоко", "Апельсин", "Груша"];

let hasApple = fruits.includes("Яблоко"); // true
let hasBanana = fruits.includes("Банан"); // false
```

- **reverse()** — этот метод меняет порядок следования элементов в массиве на обратный. Первый элемент становится последним, а последний — первым.

```
let words = ["!", "JavaScript", "люблю", "Я"];

words.reverse();

console.log(words); // ["Я", "люблю", "JavaScript", "!"]
```

- **find()** — Этот метод принимает функцию в качестве аргумента и в дальнейшем применяет ее к массиву. Он возвращает

значение элемента, найденного в массиве, если элемент удовлетворяет условию проверяющей функции. В противном случае оно возвращается со значением `undefined`.

```
let users = [
  { id: 1, name: "Yury" },
  { id: 2, name: "John" },
  { id: 3, name: "Anna" },
];

// Находим пользователя по имени
let yury = users.find(function (user) {
  return user.name === "Yury";
});

// Находим пользователя по идентификатору
let anna = users.find(function (user) {
  return user.id === 3;
});
```

- **filter()** — этот метод принимает функцию в качестве параметра и возвращает новый массив, содержащий все элементы массива, для которого функция фильтрации вернула значение `true`.

```
let users = [
  { id: 1, name: "Yury", sex: "male" },
  { id: 2, name: "John", sex: "male" },
  { id: 3, name: "Anna", sex: "female" },
];

// Находим только мужчин
let onlyMans = users.filter(function (user) {
  return user.sex === "male";
});
```

Это далеко не все существующие методы для работы с массива, но остальные используются реже и поэтому их можно [изучить позднее](#)

Object.keys, values, entries

На предыдущем занятии мы изучили объекты и даже рассмотрели отдельный цикл `for..in`, который позволяет перебирать все свойства объекта. Но в некоторых ситуациях гораздо удобнее работать с объектами, переводя их части в массивы и используя для этого методы массивов.

- **Object.keys(obj)** — возвращает массив ключей любого объекта

```
let user = {  
  name: "Yury",  
  age: 26,  
  sex: "male",  
};  
  
let keys = Object.keys(user); // ['name', 'age', 'sex']
```

- **Object.values(obj)** — возвращает массив значений любого объекта

```
let user = {  
  name: "Yury",  
  age: 26,  
  sex: "male",  
};  
  
let values = Object.values(user); // ['Yury', 26, 'male']
```

- **Object.entries(obj)** — возвращает массив пар [ключ, значение]

```
let user = {  
  name: "Yury",  
  age: 26,  
};  
  
// [['name', 'Yury'], ['age', 26]]  
let entries = Object.entries(user);
```

В некоторых случаях эти возможности могут помочь в выполнении определенных задач. Например, в одном из домашних заданий необходимо было написать функцию `isEmpty(obj)`, которая принимает любой объект и должна возвращать `true` если у него нет свойств. Одно из возможных решений может быть таким:

```
function isEmpty(obj) {  
  // Если кол-во ключей в объекте равно нулю, значит он пустой  
  return Object.keys(obj).length === 0;  
}
```

Map и Set

Сейчас мы знаем о следующих сложных структурах данных:

- Объекты для хранения именованных коллекций.
- Массивы для хранения упорядоченных коллекций.

Но этого не всегда достаточно для решения повседневных задач. Поэтому также существуют `Map` и `Set`. В данном занятии мы лишь поверхностно познакомимся с ними и не будем подробно разбирать работу с ними, т.к. в реальной практике они используются гораздо реже объектов и массивов.

Map

[Map](#) – это коллекция ключ/значение, как и `Object`. Но основное отличие в том, что `Map` позволяет использовать ключи любого типа.

Методы и свойства:

- [new Map\(\)](#) – создаёт коллекцию.
- [map.set\(key, value\)](#) – записывает по ключу `key` значение `value`.
- [map.get\(key\)](#) – возвращает значение по ключу или `undefined`, если ключ `key` отсутствует.
- [map.has\(key\)](#) – возвращает `true`, если ключ `key` присутствует в коллекции, иначе `false`.

- [map.delete\(key\)](#) – удаляет элемент по ключу key.
- [map.clear\(\)](#) – очищает коллекцию от всех элементов.
- [map.size](#) – возвращает текущее количество элементов.

Например:

```
let map = new Map();

map.set("1", "str1"); // строка в качестве ключа
map.set(1, "num1"); // цифра как ключ
map.set(true, "bool1"); // булево значение как ключ

// помните, обычный объект Object приводит ключи к строкам?
// Map сохраняет тип ключей, так что в этом случае сохранится
// 2 разных значения:
alert(map.get(1)); // "num1"
alert(map.get("1")); // "str1"

alert(map.size); // 3
```

Set

Объект [Set](#) – это особый вид коллекции: «множество» значений (без ключей), где каждое значение может появляться только один раз.

Его основные методы это:

- [new Set\(iterable\)](#) – создаёт Set, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый Set.
- [set.add\(value\)](#) – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект set.
- [set.delete\(value\)](#) – удаляет значение, возвращает true, если value было в множестве на момент вызова, иначе false.
- [set.has\(value\)](#) – возвращает true, если значение присутствует в множестве, иначе false.

- `set.clear()` – удаляет все имеющиеся значения.
- `set.size` – возвращает количество элементов в множестве.

Основная «изюминка» – это то, что при повторных вызовах `set.add()` с одним и тем же значением ничего не происходит, за счёт этого как раз и получается, что каждое значение появляется один раз.

Например, мы ожидаем посетителей, и нам необходимо составить их список. Но повторные визиты не должны приводить к дубликатам. Каждый посетитель должен появиться в списке только один раз.

```
let set = new Set();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

// считаем гостей, некоторые приходят несколько раз
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);

// set хранит только 3 уникальных значения
alert(set.size); // 3
```