4 занятие. Объекты. DOM API

Теоретическая часть

- Понятие объекта
- Свойства и методы
- Глобальный объект window
- Дерево элементов
- Поиск, изменение объектов в DOM
- Взаимодействие с веб страницей
- Создание элементов
- Шаблоны

Статьи:

Массивы и объекты в JavaScript как книги и газеты

Объекты: основы
Объекты
Object - JavaScript | MDN

Руководство по DOM - DOM | MDN
Console - Интерфейсы веб API | MDN

Практика по объектам

Практика по DOM

Понятие объекта

[Массивы и объекты в JavaScript как книги и газеты]

Объект — тип данных, который хранит в себе информацию в виде пар «ключ-значение».

Используются для хранения коллекций различных значений и более сложных сущностей.

Создание:

```
let user = new Object(); // синтаксис "конструктор объекта"

let man= {}; // литерал объекта


var man= {
// ключ          //значение
name:            'Фёдор', // свойство

 age:            28
};
```

Ключ должен быть именован по правилам именования переменных - без пробелов, не начинается с цифры и не содержит специальные символы, кроме $ и _. Но так как есть 2 варианта обращения к элементу, то может быть и не так.

```
let man = {
  name: "John",
  age: 30,
  "likes birds": true  // имя свойства из нескольких слов должно быть в кавычках
};

alert(man["likes birds"]); // true
```

Несколько правил синтаксиса:

- Ключ обособляется от значения двоеточием.
- Пары «ключ-значение» отделяются друг от друга запятыми.
- Значениями могут быть данные любого типа (число, строка, массив и так далее).

Доступ к элементам объекта происходит через точку или через квадратные скобки:

```
console.log(man.name);    // точечная нотация  -   объект.ключ
```

```
console.log(man['age']);    // скобочная нотация  -   объект[ключ]
```

Переопределение свойств

```
let man = {
  name: "John",
  age: 30,
  "likes birds": true  // имя свойства из нескольких слов должно быть в кавычках
};

man.age++;
console.log(man['age']);    //31
```

В другие переменные через оператор присваивания объекты передаются по ссылке

Задание:

1. Объявите переменную book и назначьте ей объект с именем (name) 'Программирование на Javascript' и типом (type) 'Book'

2. Напишите функцию isACat, которая определяет, является ли переданный объект котом. Объект является настоящим котом, если его имя — 'Garphield' или ему нравится молоко. Имя хранится в поле name. Что нравится объекту, хранится в строке, в поле likes. Объекту нравится молоко, если содержимое likes равно строке 'milk'

Свойства и методы

Кроме свойств хранящих значения, объект может иметь **методы** — **свойства-функции**.

Они вызываются так же, как и любые другие функции, через круглые скобки, а обращаемся мы к методам, как и к свойствам объекта.

Вызов метода записывается так: `объект.метод()`

Функции, которые что-то возвращают, называются **геттерами** и начинаются со слова **get**, а те, что что-то записывают - **сеттерами** и начинаются со слова **set**

```
let man= {
  name: "John",
  age: 30,

//метод:
sayHello: function() {
        alert("Hello " + this.name);
        }
};
```

Вызов метода: `man.sayHello();  // функция выполнится`

Ключевое слово **this** указывает на объект, на котором была вызвана функция (метод).

Вместо **объект.свойство** используется **this.свойство**.

Объект, на который указывает this называется **контекстом вызова**.

Важная деталь: пока функция не вызвана, this не содержит никакого значения, контекст появляется только в момент вызова функции

Задание: Написать функцию-конструктор объекта. На вход функции поступает 2 параметра - имя и возраст пользователя. А возвращает функция объект вида:

```
man = {
  name: "John",
  age: 30,
};
```

Добавить в конструктор метод, показывающий строку вида: “Привет, я Джон, мне 30 лет”

Подсказка: https://learn.javascript.ru/object#svoystvo-iz-peremennoy

---

Дополнительно:
Проверка существования свойства, оператор «in»

Словарь, мап или ассоциативный массив

```
var catsFavoriteFood = {
  Кекс: 'рыба',
  Рудольф: 'котлета',
  Снежок: 'сметана'
};

var printFavoriteFood = function (name) {
  return 'Моя любимая еда — ' + catsFavoriteFood[name];
};

console.log(printFavoriteFood('Снежок')); // Выведет: Моя любимая еда — сметана
```

В объект можно записать не характеристику вида name: 'Кекс', а соотношение (имени кота и лакомства). Такие объекты называют **словарями, мапами или ассоциативными массивами**. Они удобны в использовании и позволяют писать код чище и проще.

Задание: преобразовать в мапы данную информацию:

Процессор 'i5'	5000
Процессор 'i7'	10000
Дисплей 13 дюймов	5000
Дисплей 15 дюймов	10000
Оперативная память 8 Гб	3000
Оперативная память 16 Гб	7000

var processorPrice = {
  }
	var displayPrice = {
	  }
		var memoryPrice = {
		  }

Цикл «for…in»

Для перебора всех свойств объекта используется цикл for..in

```
let user = {
  name: "John",
  age: 30,
  isAdmin: true
};

for (let key in user) {
  // ключи
  alert( key );  // name, age, isAdmin
  // значения ключей
  alert( user[key] ); // John, 30, true
}
```

Удаление свойства: delete obj.prop

Задача:
Напишите функцию isEmpty(obj), которая возвращает true, если у объекта нет свойств, иначе false.

Должно работать так:

```
let schedule = {};
alert( isEmpty(schedule) ); // true
schedule["8:30"] = "get up";
alert( isEmpty(schedule) ); // false
```

Задача:
Есть объект, в котором хранятся зарплаты нашей команды:

```
let salaries = {
  John: 100,
  Ann: 160,
  Pete: 130
}
```

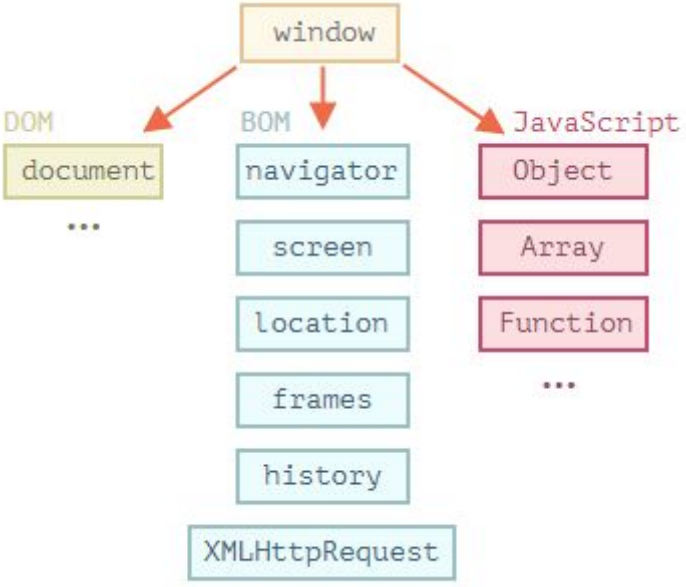
Напишите код для суммирования всех зарплат и сохраните результат в переменной sum. Должно получиться 390. Если объект salaries пуст, то результат должен быть 0.

DOM API

[Браузерное окружение, спецификации]
[Руководство по DOM]
[Манипуляции с DOM на чистом JS]
[Выразительный JavaScript: Document Object Model (объектная модель документа)]
[Об объектной модели документа | MDN]

Глобальный объект window

JavaScript в браузере:



Корневой объект **window** выступает в 2 ролях:

1. глобальный объект для JavaScript-кода
2. представляет собой окно браузера и располагает методами для управления им.

DOM — работа с элементами страницы (в том числе SVG-графикой)
BOM — работа с браузером

Document Object Model, сокращённо DOM – *объектная модель документа*, которая представляет все содержимое страницы в виде объектов, которые можно менять.

Объект **document** – основная «входная точка». С его помощью мы можем что-то создавать или менять на странице.

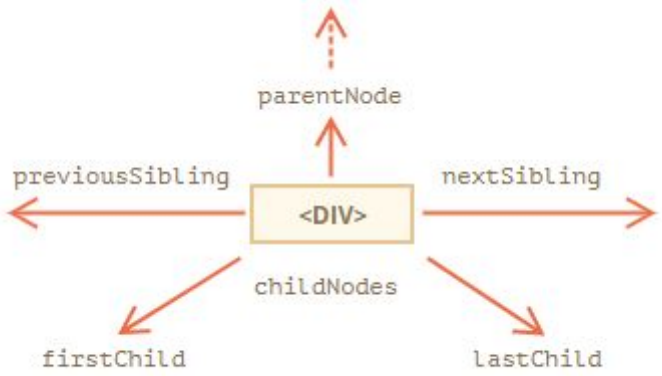
Объектная модель браузера (**Browser Object Model**, BOM) – это дополнительные объекты, предоставляемые браузером, чтобы работать со всем, кроме документа.

DOM Living Standard на https://dom.spec.whatwg.org

Дерево элементов

JavaScript особым образом воспринимает разметку: элементы здесь не строки, которые мы пишем в HTML-файлах, а объекты.

Дерево
– узел(node) — любой элемент дерева:
  – родитель (parent) — элемент, из которого растет узел
  – дети (child) — элементы, которые растут из узла
– корень — элемент из которого растет дерево, элемент без родителей
– лист — элемент дерева, который не имеет детей



Дерево может иметь только один корень. У каждого элемента должно быть не более одного родителя.

Древовидная структура JS объекта:

```
var document = {
        html: {
                head: {
                        title: 'О лосях'
                },
                body: {
                        h1: 'Вся правда о лосях',
                        a: {
                                href: 'index.html',
                                text: 'Link text'
                        }
                }
        }
};
```

методы DOM API - это группа методов, которые позволяют взаимодействовать с какой-то частью программы или интерфейса.

В случае с DOM API это все методы, которые позволяют что-то делать с DOM-элементами.

Поиск, изменение объектов в DOM

Поиск DOM-элементов

- **поиск по ID, тегу или названию класса**

  element = document.**getElementById**(id);
  - находит первый элемент с id

  elements = document.**getElementsByClassName**(names);
  - находит множество (коллекцию) элементов с классом

  elements = document.**getElementsByName**(name);
  - находит коллекцию по указанному name

  elements = document.**getElementsByTagName**(name);
  - находит коллекцию по указанному тегу

- **поиск по CSS-селектору**

  element = document.**querySelector**(selectors);
  - возвращает первый элемент документа, который соответствует указанному селектору или группе селекторов.

  elementList = document.**querySelectorAll**(selectors);
  - возвращает список, содержащий все найденные элементы документа, которые соответствуют указанному селектору.

DOM-коллекция, полученная через querySelectorAll похожа на массив. Это действительно так, она похожа, но им не является. Поэтому ещё одно название таких коллекций — псевдомассив. Перебрать коллекцию можно классическим циклом for

---

Удаление элемента

Удалять элементы со страницы можно разными способами, один из самых простых — вызов метода remove на элементе, который нужно удалить.

```
element.remove();
```

Метод из примера выше удалит element из DOM

---

innerHTML

Свойство интерфейса innerHTML устанавливает или получает HTML разметку дочерних элементов.

```
element.innerHTML = htmlString;
```

Установка значения innerHTML удаляет всё содержимое элемента и заменяет его на узлы, которые были разобраны как HTML, указанными в строке htmlString.

```
document.body.innerHTML = "";    // Заменяет содержимое body на пустую строку
```

---

textContent

Свойство textContent хранит в себе текстовое содержимое элемента.

let paragraph = document.querySelector('p');

console.log(paragraph.textContent);

Свойство textContent предназначено только для текста, если записать туда HTML-теги, браузер их не поймёт.

Новое значение textContent переписывает всё содержимое элемента

Изменение атрибутов элементов

Значением атрибута можно управлять с помощью одноимённого свойства DOM-элемента:

```
var picture = document.createElement('img');
picture.src = 'images/picture.jpg';
```

Таким же образом добавим изображению альтернативный текст, то есть описание фотографии.

```
picture.alt = 'Непотопляемая селфи-палка';
```

Названия атрибутов тегов и свойств DOM-элементов часто (но не всегда) совпадают.

Создание элемента .createElement и .append

document.createElement

```
var div = document.createElement('div');
```

Создаёт новый элемент с заданным тегом.

.append

```
элемент-родитель.append(добавляемый-элемент);
```

Метод .append добавляет набор объектов в конец (после последнего потомка) родителя.

```
var div=document.createElement("div");
var p=document.createElement("p");
p.innerHTML="qweqe";
div.append(p);
document.body.append(div);
```

Метод append не копирует элементы, а **перемещает**. Если указать в скобках элемент, который уже есть в разметке, этот элемент исчезнет со своего прежнего места и появится там, куда его добавил метод append. Получить таким образом несколько элементов не выйдет.

Задача: добавить циклом 10 абзацев в блок div.ten

Свойство style

элемент.style.свойствоCSS = "значение";

добавление свойств CSS:

```
let topBtn=document.querySelector(".btn-top");
topBtn.style.display="none";
```

Стили, заданные с помощью свойства style, работают так же, как если бы их указали в разметке в атрибуте style самого элемента. Они имеют больший приоритет, чем CSS-правила из файла со стилями.

Используйте style только в тех случаях, когда с помощью классов задачу решать неудобно.

Работа с классами classList

Среди свойств DOM-элементов — объект **classList**. Он содержит методы для управления классами DOM-элемента, в том числе и метод add(). С его помощью мы можем указать, какой класс хотим добавить элементу.

элемент**.classList.remove**('класс'); - удалить класс

элемент.**classList.add**('класс'); - добавить класс

элемент.**classList.toggle**('класс'); - переключатель класса

Обратите внимание, что точку перед названием класса ставить не нужно.

Чтобы проверить, есть ли у элемента класс, используем метод **classList.contains**

document.querySelector('.page').classList.remove('light-theme'); - убрать класс у элемента

Задача: есть массив вида:

```
var assortmentData = [
  {
    inStock: true,
    isHit: false
  },
  {
    inStock: false,
    isHit: false
  },
  {
    inStock: true,
    isHit: true
  },
  {
    inStock: true,
    isHit: false
  },
  {
    inStock: false,
    isHit: false
  }
];
```

с информацией о товаре. Повесить соответствующие классы на товары массива:
inStock: Товар в наличии — good--available. Недоступный товар — good--unavailable.
isHit: Хит продаж — good--hit.

Свойство children, дочерние элементы

querySelectorAll находит необходимые элементы один раз, фиксирует их и всё. Эта запись остаётся статичной и изменения в DOM на неё никак не влияют. Можно сказать, что querySelectorAll работает, как любая переменная, в которую мы записали какое-нибудь значение. Пока мы не переопределим переменную, в ней так и будет находиться то значение, которое мы в неё записали, независимо от того, что происходит в коде.

Поэтому такая коллекция называется **статичной**.

Кроме статичных существуют живые коллекции элементов, их ещё называют **динамическими**. Динамические коллекции реагируют на изменения в DOM. Если один из элементов коллекции будет удалён из DOM, то он пропадёт и из коллекции. Есть

несколько способов с помощью которых можно получить живую коллекцию, один из них **children**. Он вызывается на родительском элементе и собирает все дочерние элементы в динамическую коллекцию. Синтаксис такой:

```
parentElement.children;
```

```
var elList = elementNode.children;
```

В результате, elList - живая коллекция, состоящая из дочерних элементов узла elementNode, и если у узла детей нет, она будет пустой. Определить это можно, обратившись к свойству length, которое содержит в себе количество элементов в коллекции.

ParentNode.firstElementChild

- возвращает первый дочерний элемент объекта (Element) или null если дочерних элементов нет.

```
var childNode = elementNode.firstElementChild;
```

ParentNode.lastElementChild

- возвращает последний дочерний элемент объекта или null если нет дочерних элементов.

```
var element = node.lastElementChild;
```

Задача: Что выведет данный код?

```
<ul id="foo">
        <li>First (1)</li>
        <li>Second (2)</li>
        <li>Third (3)</li>
</ul>
<script>
        var foo = document.getElementById('foo'); // yields: Third (3)
         console.log(foo.lastElementChild.textContent);
</script>
```

Задача: пользователь вводит строку вроде “red, green, blue”. Нужно найти цвета в этой строке и создать на странице блоки этих цветов.

Шаблоны

Было бы удобно, если бы вся необходимая разметка для будущих элементов уже где-то хранилась. Нам бы оставалось только подправить содержимое под каждый элемент. И это можно сделать с помощью тега **template**.

Он хранит в себе шаблон для будущих элементов. Тег template находится там же, где и вся разметка сайта, только его содержимое не отображается на странице.

Чтобы получить template в JavaScript, его можно найти по названию тега, например, через querySelector('template'). У этого способа есть минус — шаблонов на странице может быть много. Обычно каждому тегу template дают уникальное название и записывают в атрибут id (идентификатор). Значения этого атрибута не могут повторяться на одной странице. По id можно найти необходимый шаблон.

Контент тега <template>, document-fragment

Если мы хотим найти элемент в шаблоне, надо искать так:

```
var template = document.querySelector('#text-template');  // Нашли template в документе

var content = template.content;  // Получили содержимое, фрагмент

var text = content.querySelector('.text');  // Нашли нужный шаблон
```

Эту запись можно сократить. Например, записать в отдельную переменную контент, а в другую искомый шаблон.

```
var textTemplate = document.querySelector('#text-template').content;
var text = textTemplate.querySelector('.text');
```

Такая запись удобней, потому что отдельно в коде элемент template обычно не используют. Вся работа ведётся с его контентом и шаблоном, который в этом контенте хранится.

В шаблоне можно менять текст, классы, а затем добавлять элементы на страницу.

Клонирование и вставка элементов

Нельзя просто так взять один элемент и добавить его много раз на страницу. Вставка сработает только один раз.

Перед нами будет блок с карточками. Шаблон для карточки хранится в теге template.

```
<body>
  …
  <template id="element-template">
    <div class="el">
      <span></span>
    </div>
  </template>
</body>
```

cloneNode()

Шаблон вставлен на страницу один раз, этот код уже написан. Попробуем вставить этот же шаблон на страницу повторно.

Для вставки элементов на страницу мы будем использовать метод **appendChild**. Он добавляет указанные элементы в конец родительского блока.

Элемент шаблона только один, и мы не можем вставить его несколько раз в разные места страницы.

Поэтому существует **клонирование DOM-элементов**. Мы можем клонировать любые элементы, в том числе шаблоны, и вставлять эти копии на страницу сколько угодно раз.

Для это нужно использовать метод **cloneNode**. Он возвращает склонированный элемент.

Обратите внимание, у этого метода есть аргумент — булево значение. Если передать значение false, то будет скопирован сам элемент со своими классами и атрибутами, но без дочерних элементов.

```
element.cloneNode(false); // Вернёт склонированный элемент без вложенностей
```

Если при передаче **false** в cloneNode копируется элемент **без вложенностей**, то при передаче **true всё наоборот**. В таком случае клонируется сам элемент вместе со всеми вложенностями. Причём клонируются атрибуты, классы и текстовое содержимое всех вложенностей. Такое клонирование называется **глубоким**.

Лучше всегда передавать булево значение, чтобы избежать непредсказуемого поведения в программах.

```
// Контейнер для карточек
var pool = document.querySelector('.pool');

// Получаем шаблон карточки
var template = document.querySelector('#element-template').content;
var element = template.querySelector('div');

// Клонируем и наполняем элемент

  var clonedElement = element.cloneNode(true);
   clonedElement.children[0].textContent = 3;
   pool.appendChild(clonedElement);
```

Также можно использовать цикл for для многократного добавления элементов

```
for (var i = 1; i <= 10; i++) {
   var clonedElement = element.cloneNode(true);
   clonedElement.children[0].textContent = i;
   pool.appendChild(clonedElement);
 }
```

Практика по объектам

Умножаем все числовые свойства на 2

Создайте функцию multiplyNumeric(obj), которая умножает все числовые свойства объекта obj на 2.

Например:

```
// до вызова функции
let menu = {
  width: 200,
  height: 300,
  title: "My menu"
};
multiplyNumeric(menu);

// после вызова функции
menu = {
  width: 400,
  height: 600,
  title: "My menu"
};
```

Обратите внимание, что multiplyNumeric не нужно ничего возвращать. Следует напрямую изменять объект.

P.S. Используйте typeof для проверки, что значение свойства числовое.

Создайте калькулятор

Создайте объект calculator (калькулятор) с тремя методами:

read() (читать) запрашивает два значения и сохраняет их как свойства объекта.
sum() (суммировать) возвращает сумму сохранённых значений.
mul() (умножить) перемножает сохранённые значения и возвращает результат.

```
let calculator = {
  // ... ваш код ...
};

calculator.read();
alert( calculator.sum() );
alert( calculator.mul() );
```

Работа программы: Окно ввода первого числа, окно ввода второго числа, окно “Сумма равна: ...”, окно “Произведение равно: ...”

Дом

Допиши конфигуратор. Есть объект house и несколько его свойств: rooms (количество комнат), floors (этажи), material (материал для стен), coefficient (средняя площадь каждой комнаты).

Есть мап materialPrice, в которой записана стоимость каждого возможного материала для строительства.

Добавь в объект два метода: calculateSquare, который будет возвращать площадь дома, и calculatePrice, который будет возвращать стоимость строительства.

Площадь считай так: умножь количество комнат на коэффициент и число этажей в доме. Цена строительства — произведение площади и стоимости материала дома.

Дано:

```
var materialPrice = {
  'wood': 1000,
  'stone': 1500,
  'brick': 2000
};

var house = {
  rooms: 10,
  floors: 5,
  material: 'wood',
  coefficient: 10.5
}
```

```
Первый тест. Исходное значение:
{"rooms":1,"floors":1,"material":"stone","coefficient":1}
Ожидаю результат площадь: 1, стоимость: 1500.

Второй тест. Исходное значение:
{"rooms":6,"floors":8,"material":"brick","coefficient":10}
Ожидаю результат площадь: 480, стоимость: 960000.

Третий тест. Исходное значение:
{"rooms":3,"floors":6,"material":"wood","coefficient":5.5}
Ожидаю результат площадь: 99, стоимость: 99000.
```

Переводчик

Напиши программу-переводчик. Создай функцию translate с двумя параметрами. Первый параметр — строка со словом на русском языке, которое нужно перевести на английский. Второй параметр — объект с данными, в котором хранится перевод слов. Функция должна возвращать строку вида: понедельник по-английски: monday.

Дано:

```
var daysOfWeek = {
  'понедельник': 'monday',
  'вторник': 'tuesday',
  'среда': 'wednesday',
  'четверг': 'thursday',
  'пятница': 'friday',
  'суббота': 'saturday',
  'воскресенье': 'sunday'
};
```

Золотой мяч

Написать программу, которая подсчитает полезность и результативность игроков на основе их статистики. Оформи код в виде функции getStatistics с одним параметром — массивом игроков.

Каждый футболист в этом массиве описывается объектом с тремя полями: имя (свойство name), забитые голы (свойство goals) и голевые пасы (свойство passes). Функция должна возвращать этот же массив, в котором каждому игроку добавлены ещё два поля: коэффициент полезности (свойство coefficient) и результативность (свойство percent).

Коэффициент полезности считается так: умножаем голы игрока на 2 (потому что я считаю, что голы важнее всего) и прибавляем к этому значению все голевые пасы футболиста.

Результативность (процент забитых мячей футболиста от результата всей команды) считаем так: находим сумму голов всех игроков и выясняем, сколько процентов от этого числа забил каждый футболист. Округляй значение с помощью Math.round().

```
Первый тест. Исходное значение:
[{"name":"Васька","goals":5,"passes":5},{"name":"Байт","goals":12,"passes":2},{"name":"Снежок","goals":2,"passes":7}]
Ожидаю результат:
[{"name":"Васька","goals":5,"passes":5,"coefficient":15,"percent":26},{"name":"Байт","goals":12,"passes":2,"coefficient":26,"percent":63},{"name":"Снежок","goals":2,"passes":7,"coefficient":11,"percent":11}]

Второй тест. Исходное значение:
[{"name":"Васька","goals":3,"passes":7},{"name":"Байт","goals":5,"passes":2},{"name":"Снежок","goals":15,"passes":2}]
Ожидаю результат:
[{"name":"Васька","goals":3,"passes":7,"coefficient":13,"percent":13},{"name":"Байт","goals":5,"passes":2,"coefficient":12,"percent":22},{"name":"Снежок","goals":15,"passes":2,"coefficient":32,"percent":65}]

Третий тест. Исходное значение:
[{"name":"Васька","goals":3,"passes":2},{"name":"Байт","goals":10,"passes":1},{"name":"Снежок","goals":2,"passes":14}]
Ожидаю результат:
```

[{"name":"Васька","goals":3,"passes":2,"coefficient":8,"percent":20},{"name":"Байт","goals":10,"passes":1,"coefficient":21,"percent":67},{"name":"Снежок","goals":2,"passes":14,"coefficient":18,"percent":13}]

Повторы слов

В этой задаче вам нужно проанализировать данные — вычислить повторы каждого слова.

Создайте функцию getRepeats с одним параметром. В этот параметр будет приходить массив данных.

Функция должна возвращать объект, в котором указано сколько раз каждое слово встречается в массиве. Объект должен быть такого вида:
{
  'одно слово': 1,
  'другое слово': 2,
  'ещё одно слово': 5
}
Обратите внимание, что счёт начинается не с нуля, а с единицы. Если вы встречаете слово в массиве в первый раз, значит надо записать, что слово встречается один раз, а не ноль.

Первый тест. Массив данных:
["Василий","Пётр","Иннокентий","Пётр","Иван","Василий"]
Ожидаемый результат:
{"Василий":2,"Пётр":2,"Иннокентий":1,"Иван":1}

Второй тест. Массив данных:
["привет","пока","прощай","пока","здравствуйте","прощай"]
Ожидаемый результат:
{"привет":1,"пока":2,"прощай":2,"здравствуйте":1}

Третий тест. Массив данных:
["картошка","картошка","кофе","торт","салат","кофе","торт","картошка"]
Ожидаемый результат:
{"картошка":3,"кофе":2,"торт":2,"салат":1}

Четвёртый тест. Массив данных:
["работа","работа","работа","работа","работа"]
Ожидаемый результат:
{"работа":5}

Сжатие массивов

В этом задании вы соберёте объект с данными на основе двух массивов.
Создайте функцию getZippedArrays.

У функции должно быть два параметра. Первый — массив с названиями ключей.
Второй — массив со значениями этих ключей.

Например:

первый массив ["Имя", "Возраст", "Цвет глаз", "Знает JS"]
второй массив ["Петя", 28, "карие", true]

Результат:
User {
"Имя": "Петя" ,
"Возраст": 28,
"Цвет глаз": "карие",
"Знает JS": true
}

Сортировка объектов

В этом задании вам нужно написать сортировку в массиве объектов. Вы можете использовать любой алгоритм сортировки.

Создайте функцию getSortedArray. У неё должно быть два параметра. Первый – массив, который нужно отсортировать. Второй – имя ключа в объектах. Именно по значению этого ключа нужно будет делать сортировку.

Функция должна возвращать отсортированный массив объектов. Значения в массиве должны увеличиваться от меньшего к большему.

Примерно так будет выглядеть результат работы вашей программы:

```
// Массив, который надо отсортировать
// Сортировать будем по значению в ключе age
[
  {
    name: 'Петя',
    age: 5
  },
  {
    name: 'Лёля',
    age: 2
  },
  {
    name: 'Сима',
    age: 3
  }
];

// Отсортированный массив
[
```

```
  {
    name: 'Лёля',
    age: 2
  },
  {
    name: 'Сима',
    age: 3
  },
  {
    name: 'Петя',
    age: 5
  }
];
```

Собираем массив объектов

В этом задании вам нужно на основании массивов с данными собрать массив объектов.

Создайте функцию getData. У неё должно быть два параметра. Первый параметр — массив с ключами. Второй — массив с массивами данных.

Функция должна собрать объект для каждого массива значений. И каждый из этих объектов должен быть записан в массив данных. Именно этот массив должна вернуть функция getData.

Каждому элементу из массива ключей подходит элемент с таким же индексом в массиве значений.

Есть один нюанс: значений может оказаться больше или меньше, чем ключей.
Если значений не хватает, то создавать пустой ключ не надо.
А если значений больше, то их не нужно включать в объект — для них нет ключей.

Примерно так должен выглядеть результат работы вашей программы:

```
// Массив ключей
['имя', 'любимый цвет', 'любимое блюдо'];
// Массив значений
[
  ['Василий', 'красный', 'борщ'],
  ['Мария'],
  ['Иннокентий', 'жёлтый', 'пельмени', '18', 'Азовское']
];

// Готовый массив объектов
[
```

```
  {
    'имя': 'Василий',
    'любимый цвет': 'красный',
    'любимое блюдо': 'борщ'
  },
  {
    'имя': 'Мария'
  },
  {
    'имя': 'Иннокентий',
    'любимый цвет': 'жёлтый',
    'любимое блюдо': 'пельмени'
  }
];
```

```
Первый тест. Массив ключей:
["sea","country","city"]
Массив значений:
[["Балтийское","Эстония","Силламяэ"],["Охотское","Россия","Охотск"],["Жёлтое","Китай","Бэйдайхэ"]]
Ожидаемый результат:
[{"sea":"Балтийское","country":"Эстония","city":"Силламяэ"},{"sea":"Охотское","country":"Россия","city":"Охотск"},{"sea":"Жёлтое","country":"Китай","city":"Бэйдайхэ"}]
Второй тест. Массив ключей:
["name","growth","weight","age"]
Массив значений:
[["Пётр","165","70"],["Василий","170"],["Светлана"]]
Ожидаемый результат:
[{"name":"Пётр","growth":"165","weight":"70"},{"name":"Василий","growth":"170"},{"name":"Светлана"}]
Третий тест. Массив ключей:
["performer","album","song"]
Массив значений:
[["Robbie Williams","The Heavy Entertainment Show","Love My Life"],["Billie Eilish","When We All Fall Asleep, Where Do We Go?","Bag Guy","17","Los Angeles"],["Монеточка","Раскраски для взрослых","90","20","Екатеринбург"]]
Ожидаемый результат:
[{"performer":"Robbie Williams","album":"The Heavy Entertainment Show","song":"Love My Life"},{"performer":"Billie Eilish","album":"When We All Fall Asleep, Where Do We Go?","song":"Bag Guy"},{"performer":"Монеточка","album":"Раскраски для взрослых","song":"90"}]
```

Практика по DOM

Верстка: https://codepen.io/lipa88/pen/VwjLpgq

Объявите переменную specialProduct и запишите в неё второй DOM-элемент из списка товаров.

Создайте переменную unavailableProduct и запишите в неё последний DOM-элемент из списка товаров.

product--available для товара в наличии;
product--unavailable соответствует товару, которого в наличии нет;
product--special для спецпредложения.

Добавьте specialProduct класс product--special.
Добавьте unavailableProduct класс product--unavailable.

Есть интернет-магазин с готовой вёрсткой https://codepen.io/lipa88/pen/VwjLpgq. Нужно показывать в интерфейсе актуальную информацию о товарах: спецпредложения и наличие на складе.

Данные приходят в виде массива объектов catalogData. Каждый объект соответствует одному товару и содержит свойства isAvailable (в наличии товар или нет) и isSpecial (является ли товар спецпредложением или нет).

Для каждого состояния товара есть соответствующий класс:

product--available для товара в наличии;
product--unavailable соответствует товару, которого в наличии нет;
product--special для спецпредложения.

Массив данных о каждом товаре:

```
var catalogData = [
 {
   isAvailable: true,
   isSpecial: false
 },
 {
   isAvailable: false,
   isSpecial: false
```

```
  },
  {
    isAvailable: true,
    isSpecial: true
  },
  {
    isAvailable: true,
    isSpecial: false
  },
  {
    isAvailable: false,
    isSpecial: false
  }
];
```

Алгоритм:
После массива **catalogData** объявите функцию **updateCards** с параметром products.

Ниже вызовите функцию updateCards с аргументом catalogData.

В теле функции создайте переменную elements и запишите в неё список DOM-элементов, найденных по селектору .product с помощью querySelectorAll.

В теле функции updateCards:

1. Напишите цикл for, который перебирает список elements, увеличивая значение переменной i от 0 до длины списка.
2. Внутри цикла создайте переменную element и запишите в неё текущий DOM-элемент списка elements[i].
3. Добавьте element класс product--available.

Создайте переменную product, которая равна текущему элементу массива products, а затем выведите её в консоль.

После вывода product создайте переменную availabilityClass со значением 'product--available'.

После этой переменной добавьте проверку, что товара нет в наличии.

Если условие выполняется, переопределите значение availabilityClass на 'product--unavailable'.

В вызове метода classList.add() у текущего DOM-элемента замените строку 'product--available' на переменную availabilityClass.

В конце тела цикла добавьте ещё одну проверку, что на товар распространяется спецпредложение.

Если условие срабатывает, добавляйте текущему DOM-элементу класс product--special.

Удалите из кода все выводы в консоль.

На сайте магазина мороженого надо отображать актуальное состояние товаров: «в наличии», «нет в наличии», «хит».

https://codepen.io/lipa88/pen/XWKbMvZ?editors=1100

Данные по продуктам хранятся в массиве с объектами assortmentData, каждый объект соответствует одному товару и содержит свойства:

inStock. Если значение true — мороженое в наличии, если false — товара в наличии нет.
isHit. Если значение true — мороженое самое популярное среди покупателей.
Каждому состоянию товара соответствует специальный класс:

Товар в наличии — good--available.
Недоступный товар — good--unavailable.
Хит продаж — good--hit.
Оформи код в виде функции updateCards, которая принимает на вход массив с данными. Вызови её, передав assortmentData.

```
var assortmentData = [
 {
  inStock: true,
  isHit: false
 },
 {
  inStock: false,
  isHit: false
 },
 {
  inStock: true,
  isHit: true
 },
 {
  inStock: true,
  isHit: false
 },
 {
  inStock: false,
  isHit: false
 }];
```