**ТЕЗИСЫ ПО ПОСЛЕДНИМ УРОКАМ КУРСА WEB-разработчик 2020 (Udemy)**

SCSS обратно совместим с CSS

Нижнее подчёркивание перед названием файла нужно для того, чтобы он не компилировался (содержимое файла импортируется в другие SCSS файлы).

src – папка с исходниками, где лежат рабочие файлы (черновые, не минимизированные)

dist – папка для выгрузки на хостинг, внутри которой минифицированные картинки, сжатые скрипты и стили и т.д.

\* - одна звезда – любой файл заданного формата/расширения

\*\* - файл из любой папки(файлы из всех вложенных папок)

**ТЕЗИСЫ ИЗ КНИГИ JS FOR KIDS**

Запуск метода в программировании называется вызовом метода

Скобки и то, что в сразу задано в скобках – литерал массива или объекта

Когда JS автоматически преобразует значение к другому типу, называется неявным приведением типа

Декрементировать – уменьшать

Длинная запись функции в JS называется функциональным выражением, а короткая – объявлением функции.

DOM – Document Object Model объектная модель документа используется браузерами для структурирования страниц и их элементов.

ООП – это способ проектирования и написания кода, когда все важные части программы являются объектами.

Помимо строк, чисел и булевых значений, в свойствах объекта можно хранить функции – тогда эти свойства называют методами.

Ключевое слово *this* можно использовать в теле метода, чтобы обратиться к объекту, для которого метод вызывается.

Чтобы использовать один и тот же код метода с разными объектами, достаточно добавить его в виде свойства каждому из этих объектов.

В JS Конструктор – это функция, которая создает объекты, давая им набор заранее определенных свойств и методов. Для вызова конструктора используется ключевое слово *new*, а следом имя конструктора и скобки. Конструкторы называют с Заглавной буквы.

Прототипы JS – это механизм, который упрощает использование общей функциональности (т.е. методов) разными объектами. У всех конструкторов есть свойство prototype, к которому можно добавлять методы; любой метод, добавленный к этому свойству будет доступен всем объектам, которые созданы с помощью этого конструктора.

Контекст рисования(getContext )– это JavaScript объект, обладающий методами и свойствами, при помощи которых можно рисовать на «холсте»(canvas).

Инвертировать значение – умножить на -1.

Информация о том, какая клавиша на клавиатуре была нажата хранится в свойстве keyCode объекта event (event.keyCode).

Каждая клавиша соответствует уникальному числовому коду.

В HTML и CSS в основном выполняются декларативные операции.

В JS любая переменная, которая содержит (непустую) строку, считается истинной, а переменная, которой еще не было присвоено значение, считается ложной.

**КОНСПЕКТ ПО КУРСУ «ПОЛНЫЙ КУРС ПО JavaScript…» 2020 (ОБНОВЛЁННЫЙ Udemy)**

ПОДКЛЮЧЕНИЕ JS К СТРАНИЦЕ

1. Непосредственно в html файл:

- в <head>, тогда скрипт загрузится до загрузки страницы; способ делает загрузки страницы более долгой, интерактивные элементы скорее всего работать не будут, нужно чётко понимать необходимость подключения скрипта именно в <head>

- в конце <body>, тогда скрипт загрузится после загрузки страницы, используется редко

2) В качестве отдельного файла перед закрывающим тэгом </body>:

 <script src="script.js"></script>

В проекте создаётся папка js/script.js

1. Атрибуты тегов, запускающие какую либо функцию, прим <div onclick = “…”>Hello World</div> считается плохой практикой.

ПЕРЕМЕННЫЕ

Переменные можно объявлять через запятую с ; в конце, прим. let a = 10, b = 20;

Если переменная объявлена с помощью var, она видна везде, даже до того как она была объявлена (скорее всего считывается при первом проходе кода вместе с объявлением функций). Такое поведение называется хостинг или всплытие переменной

Если переменная объявлена с помощью let, она создаётся только тогда, когда код до неё доходит (скорее всего переменная считывается браузером при втором проходе последовательно, вместе с остальным кодом). Смысл объявления с помощью let – код изначально работает быстрее (и экономится ОЗУ), т.к. заранее не хранит множество переменных. let существует/видна только в блоке кода, ограниченного {}.

Если переменная объявляется с помощью const, создаётся константа, которую нельзя поменять. const, как и let не видна в коде до её инициализации, существует/видна только в блоке кода, ограниченного {}. Хорошим тоном считается использование const везде, где это возможно. НО как таковых констант в JS не существует, пример – значение свойств и методов объекта {} заданные через const можно переопределять, в.т.ч. добавлять новые и удалять существующие.

; ставится после объявления переменных и после окончания логического блока.

ТИПЫ ДАННЫХ

Учитывая новый стандарт, в JS существует 8 типов данных:

Число – целые и дробные, infinity(можно получить при делении на ноль), NaN (операции без мат. логики, прим. Число поделить на строку)

Строка – помещается в кавычки “ ” ‘’ ` `

Символ – новый тип данных Symbol(); используется очень редко.

Логическое(Булево) значение – false true

null – когда чего-то в коде просто не существует

undefined – когда что-либо в коде уже существует, но ещё не имеет значения

объект – комплексный тип данных, в JS всё является объектом массив (частный случай объекта), функции, объект даты, регулярные выражения, ошибки.

BigInt – большие числа, т.к. в JS есть ограничения по работе с обычными числами – нельзя задать число, которое больше чем 253.

ОПЕРАТОРЫ

Синтаксис prompt(“Есть ли вам 18 ?”, “Да”) – Да – заполнитель окна(значение по умолчанию default). prompt всегда даёт строку, но если перед ним поставить + унарный плюс , он выдаст числовой тип данных (если перед значением представленным строкой поставить + унарный плюс, то оно превратится в числовой тип данных).

Унарный плюс – это + который ставится перед определённым аргументом, унарный потому что использует только один аргумент для своей работы. Пример +”5” для преобразования строки в число 5.

+ - конкатенация

++ -инкремент (увеличение на единицу)

-- - декремент (уменьшение на единицу)

Префиксная форма инкремента и декремента возвращает изменённое значение, а постфиксная – старое. Прим. let a = 10; console.log(--a); вернёт 9 console.log(a--); вернёт 10.

% возвращает остаток от деления двух чисел, прим. 5%2 вернёт 1.

= - присваивание

== - проверка на равенство(с преобразованием типов при необходимости)

=== - проверка на строгое равенство (в т.ч. типов данных)

Если код пишется согласно нового стандарта ES6, в начале документа прописывается ‘use strict’; – строгий режим; при наличии этой директивы ошибки и баги/недочёты предыдущих версий в коде работать не будут. Так же иногда прописывается в начале функций.

УСЛОВИЯ

Для того чтобы избежать больших ветвлений if, используется специальная конструкция switch, модификация if которая поддерживает сразу несколько проверок и условий:

const num = 50;

switch (num) {

    case 49:

        console.log('Неверно');

        break;

    case 100:

        console.log('Неверно');

        break;

    case 50:

        console.log('Верно');

        break;

    default:

        console.log('Не в этот раз');

        break;

}

ВАЖНО! Switch всегда проводит строгое сравнение в отличии от обычных условий.

break; обязательная синтаксическая конструкция, после case, т.к. если вдруг условие выполнится, скрипт продолжит дальше проверять остальные case’ы, а с break проверка сразу прекратится.

Case’ов может быть очень много и не один из них может не выполниться, тогда для выполнения действия по умолчанию используется default, этот блок является необязательным.

ЦИКЛЫ

Реализуются в JS 3мя способами:

1. while(условие) {что-то делать} – пока условие выполняется делать что-то
2. do{что-то делать} while(условие) - сначала цикл что-то делает, а потом проверяет условие если необходимо выйти из цикла.
3. for(итератор; условие остановки цикла; шаг цикла) {что-то делать} – условие в скобках состоит из 3х аргументов, но они не обязательны.

Чтобы выйти из цикла до его завершения, используется условие с break:

for(let i = 1;i < 10; i++) {

    if (i==6){

        break;

    }

    console.log(i);

}

continue – оператор позволяет пропустить шаг заданный в условии, но при этом не прерывает цикл. На практике используется для исключения чётных/нечётных или конкретных значений

let num = 50;

for (let i = 1; i < 10; i++) {

    if (i == 6) {

        continue;

    }

    console.log(i);

}

ФУНКЦИИ

Правило: имя функции это глагол с припиской того, над чем выполняется действие(showMessage).

Переменные, объявленные внутри функции, в т.ч. параметры снаружи недоступны:

function multiply(a, b, c) {

    return a + b \* c;

}

console.log(multiply(20, 20, 20));

console.log(c);

c is not defined, a,b,c – локальные переменные.

НО глобальные переменные могут быть переопределены функцией, при УСЛОВИИ, что они не указаны в качестве параметров(=локальная переменная?) функции:

let num = 20;

function printNum() {

    console.log('Some text');

    num = 100;

}

printNum();

console.log(num);

Выдаст в консоль 100.

let num = 20;

function printNum(num) {

    console.log('Some text');

    num = 100;

}

printNum();

console.log(num);

Выдаст в консоль 20.

let num = 20;

function printNum() {

    console.log('Some text');

    let num = 100;

}

printNum();

console.log(num);

Аналогично выдаст 20, т.к. тут num локальная переменная функции.

Замыкание функции – это сама функция вместе со всеми внешними переменными, которые ей доступны.

Директива return может находиться в любом месте функции, как только выполнение доходит до этого места, функция останавливается и значение возвращается в вызвавший её код. Использование return без значения используется для немедленного выхода из функции. Код после return называется Unreacheble, т.к. никогда не будет выполнен:

function printNum() {

    console.log('Some text');

    let num = 100;

    return;

    console.log('Unreachable code');

}

НО return может быть вызван несколько раз, например для проверки условий:

function checkAge(age) {

if (age > 18) {

return true;

} else {

return confirm('А родители разрешили?');

}

МЕТОДЫ И СВОЙСТВА СТРОК И ЧИСЕЛ

Методы напрямую не меняют исходных значений.

Методы строк

Методы изменения регистра строк:

.toUpperCase() – в верхний регистр

.toLoweCase() – в нижний регистр

Метод поиска подстроки, позволяет найти часть строки и с какого индекса эта часть начинается:

const text = "Some text";

console.log(text.indexOf("text"));

Выдаст 5, т.к. часть со словом text начинается с 5го индекса. Если искать несуществующее значение, метод выдаст -1.

Методы модифицирующие(изменяющие) строки:

1. .slice(начало\_подстроки, конец\_подстроки) символ с индексом конец\_подстроки не включается в саму подстроку. Если указать только один аргумент, подстрока будет вырезана до конца. Метод принимает и отрицательные значения, тогда отсчёт индексов подстроки начинается с конца строки

const text = "Hello world";

console.log(text.slice(6, 11));

const text = "Hello world";

console.log(text.slice(-5));

Выдаёт world

1. .substring(начало\_подстроки, конец\_подстроки) отличие от .slice() в том что первый аргумент можно задавать больше чем второй, а так же он не поддерживает отрицательные значения
2. .substr(начало\_подстроки, количество\_символов)

const text = "Hello world";

console.log(text.substr(6, 5));

Выдаёт world

Методы чисел

Для работы с числами в JS существует встроенная библиотека Math и её методы.

const num = 12.5;

console.log(Math.round(num));

Округляет число до ближайшего целого, выдаст 13.

const test = "12.2 px";

console.log(parseInt(test));

parseInt() разбирает строковый аргумент(читает числа из строки) и возвращает целое число, вернёт число 12 (не строку!)

const test = "12.2 px";

console.log(parseFloat(test));

parseFloat() разбирает строковый аргумент и возвращает число с плавающей запятой, вернёт число 12.2

CALLBACK ФУНКЦИИ

Правило: если функции идут в коде одна за другой, это не значит что они выполняются в этой же последовательности, они запускаются одна за другой, но результат могу выдать в разное время.

Callback – функция, которая должна выполняться после того как другая функция завершила свою работу.

function learnJS(lang, callback) {

    console.log(`Я учу ${lang}`);

    callback();

}

learnJS('JavaScript', () => {

    console.log('Я прошел этот урок!');

});

В качестве callback функции может выступать как анонимная, так и имеющая имя функция.

function learnJS(lang, callback) {

    console.log(`Я учу ${lang}`);

    callback();

}

function done() {

    console.log('Я прошел этот урок!');

}

learnJS('JavaScript', done);

ВАЖНО! Вторым аргументов для функции learnJS передаётся функция done а не её вызов! Она передаётся вместо callback и выполнится только тогда, когда скрипт до неё дойдёт. Мы не вызываем функцию, а передаём, чтобы она была использована в нужный момент!

На практике колбэки используются постоянно (при запросах к серверу, с событиями на странице). Серверное программирование на Node.js почти полностью построено на колбэках.

МАССИВЫ И ПСЕВДОМАССИВЫ

.pop - удаляет последний элемент массива.

.push – добавляет элемент в конец массива.

.shift – удаляет элемент в начале массива.

.unshift – добавляет элемент в начало массива.

.shift и .unshift очень редко используются, т.к. после их применения нужно переиндексировать весь массив, время затраченное на это тем больше чем больше массив.

Вопрос: Как соотносятся между собой свойство .length и порядковые номера элементов массива? Ответ: При работе с массивом .length возвращает не количество существующих элементов в массиве, а индекс последнего элемента+1.

Перебор массивов:

for (let i = 0; i < arr.length; i++) {

    console.log(arr[i]);

}

Перебор массива с помощью цикла for.

for (let value of arr) {

    console.log(value);

}

for(let key in имя\_массива) – цикл перебора, возвращающий индексы элементов массива.

for(let key of имя\_массива) – цикл перебора, возвращающий (значения) элементы массива.

Перебор массива с помощью цикла for of. Работает только с массивоподобными сущностями – массив, срока, псевдомассив, map, set. Цикл так же может перебрать определённые элементы со страницы. Плюсом относительно метода .forEach, является то, что внутри цикла можно использовать ключевые слова break и continue.

Методы массивов:

.forEach(callback функция(item, index, array)) – метод для перебора массива, в зависимости от того, что нужно получить, в аргументе callback функции могут использоваться одно или несколько значений. Используется часто.

arr.forEach((item, i, arr) => {

    console.log(`${i}: ${item} внутри массива ${arr}`);

});

.split(‘, ’) – разбивает строку, превращая её в массив путём разделения после указанного символа (подстроки ‘, ’).

const str = prompt("", "");

const products = str.split(", ");

console.log(products);

.join(‘/’) – обратный метод, берёт каждый элемент массива и соединяет в строку, в аргументе принимает разделитель элементов(‘; ’).

const str = prompt("", "");

const products = str.split(", ");

console.log(products.join("; "));

.sort() – метод сортирует массив по алфавиту, используется для строковых значений, т.к. воспринимает числовые значения как строки.

const arr = [1220, 20, 35, 63, 8];

arr.sort();

console.log(arr);

Выдаст: [1220, 20, 35, 63, 8], т.к. сортировка посимвольно, а 0<1<2<3<4<5….

Для сортировки числовых значений нужно передать в аргумент callback, САМУ функцию не ВЫЗОВ! function …(a,b){return a-b}. После передачи функции, метод ориентируется на разность между двумя элементами массива, исходя из этого сортирует их по порядку.

const arr = [1220, 20, 35, 63, 8];

arr.sort(compareNum);

console.log(arr);

function compareNum(a, b) {

    return a - b;

}

Метод .sort внутри себя использует алгоритм быстрой сортировки.

Псевдомассивы получаются при работе с элементами на странице. У псевдомассива нет вышеперечисленных методов.

ОБЪЕКТЫ

JS считается объектно ориентированным языком (но точнее говорить прототипно ориентированным). Объекты в JS = ассоциативные массивы в других языках (PHP). В JS почти всё является объектом и получает свои методы через цепочку прототипов от объекта Object.

имя\_объекта.имя\_свойства = звачение\_свойства – добавление свойства объекта.

delete options.name;

delete имя\_объекта.свойство\_объекта - удаление свойства объекта.

for (let key in options) {

    console.log(`Свойство ${key} имеет значение ${options[key]}`);

}

цикл перебора свойств объекта for (let key in имя\_объекта) {console.log(key + “ : ” + имя\_объекта[key]}; (выводит пары ключ : значение). Цикл for of для объектов не работает.

for (let key in options) {

    if (typeof(options[key]) === 'object') {

        for (let i in options[key]) {

            console.log(`Свойство ${i} имеет значение ${options[key][i]}`);

        }

    } else {

        console.log(`Свойство ${key} имеет значение ${options[key]}`);

    }

}

Если внутри объекта может находиться объект, чьи свойства так же необходимо получить, то используется перебор внутри перебора. Для этого вначале свойства объекта проверяются на принадлежность к типу ‘object’, если условие верно, выполняется тело цикла с перебором свойств вложенного объекта, если нет то перебираются свойства «основного» объекта.

Т.к. у объекта нет свойства .length, чтобы сосчитать количество свойств объекта используется Приём счётчика:

let counter = 0;

for (let key in options) {

    counter++;

}

console.log(counter);

Object.keys(options)

Метод принимает объект и на его основе создаёт массив, в котором все элементы – это «ключи» находящиеся на первом уровне вложенности представленные в виде строк.

Object.keys(options).length

А у массивов уже есть свойство .length,поэтому Object.keys(имя\_объекта).length – выдаёт количество свойств объекта.

У объектов так же существуют свойства акцессоры get и set.

В ES6 появилась деструктуризация объектов(и массивов) – служит для «вытаскивания» элементов в качестве отдельных структур.

const { border, bg } = options.colors;

console.log(border);

border и bg свойства объекта colors, который находится внутри(является свойством) объекта options.

ООП

ООП – это наука о том как делать правильную архитектуру. В JS всё стоится на прототипах.

имя\_объекта\_потомка.\_\_proto\_\_ = имя\_объекта\_родителя – способ задания наследования свойств и методов одного объекта от другого.

1: const soldier = {

2:     health: 400,

3:     armor: 100

4: };

5: const john = {

6:     heath: 100

7: };

8: john.\_\_proto\_\_ = soldier;

9: console.log(john.armor);

Выдаст 100, свойство armor изначально отсутствует в объекте john, но после того как его прототипом объявляется объект soldier, john наследует это свойство от soldier.

Такой формат задания прототипа считается устаревшим. Изменение прототипа влияет на производительность кода, вместо этого рекомендуется создавать объект с нужным прототипом с помощью метода [Object.create()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/create).

В реальных проектах работает следующим образом: создаётся один большой прототип модального окна, а от этого прототипа создаётся нужное количество модальных окон, которые немного отличаются.

const soldier = {

    health: 400,

    armor: 100

};

const john = {

    heath: 100

};

Object.setPrototypeOf(john, soldier);

john.\_\_proto\_\_ = soldier;

console.log(john.armor);    //100

Тот же код в современном исполнении, используя метод .setPrototypeOf. Применяется если нужно назначить прототип в «динамике», т.е. когда объекты уже существуют.

НО! обычно прототипы задаются на этапе создания объектов

const john = Object.create(soldier);

создаётся новый объект john который будет прототипно наследоваться от soldier

Метод **Object.getPrototypeOf()** возвращает прототип (то есть, внутреннее свойство [[Prototype]]) указанного объекта.

ДИНАМИЧЕСКАЯ ТИПИЗАЦИЯ

Это возможность одного типа данных превращаться в другой (число может стать строкой, строка числом, а объект булиновым значением)

В других ЯП существует и статическая типизация(где число всегда число).

Все данные, получаемые от пользователя, это строки!

Превращение в строку:

1. String(…) устаревший метод
2. Конкатенация(сложение, +) чего либо со строкой, можно даже пустой, прим. ‘’ + 4

Пример 1 – динамическое формирование ссылок:

const num = 5;

console.log("https://vk.com/catalog" + num);

Пример 2 – динамическое формирование стилей:

const fontSize = 26 + 'px';

Превращение в число:

1. Number(…) устаревший метод
2. Унарный плюс, прим. 5 + + ‘5’ // = 10
3. С помощью метода чисел parseInt(…, 10) принимает строку и возвращает целое число, 10 – десятичная система, способ используется редко.

console.log(typeof(parseInt('15px', 10)));

Преобразование в Булево значение:

1. 0, ‘’, null, undefined, NaN – псевдоложь, всё остальное псевдоистина.

Пример:

let switcher = null;

if (switcher) {

    console.log('do something');

}

switcher = 1;

Если какого-то элемента на странице не существует, вместо него будет null, программа работает и с определённым промежутком проверяет условие if и соответственно не выполняет его(null псевдоложь), но в какой-то момент элемент появляется и помещается в switcher (1 псевдоистина), в очередной раз проверяется условие и на этот раз начинает выполняться.

1. Boolean(…) используется редко
2. !! – два знака отрицания перед любым выражением используется редко

console.log(typeof(!!'10000'));

ПОЛУЧЕНИЕ ЭЛЕМЕНТОВ СО СТРАНИЦЫ

Весь html может быть представлен обычным документом, у которого есть своя структура, и этот документ может быть представлен в виде дерева узлов, при этом узлы связаны между собой отношениями родительский – дочерний. Такая структура создаётся во время вёрстки неважно чего – лэндинга или приложения. Такой документ можно представить в виде обычного объекта, именно отсюда произошло название DOM(document object model объектная модель документа).

У DOM как и у любого объекта есть свои методы. Для обращения к DOM используется глобальный объект(сущность) document, он работает только внутри браузера.

Методы с помощью которых позволяют получить элементы со страницы можно разделить на 2 категории:

1. Методы получения элементов, которые существуют давно(устоявшиеся):

const box = document.getElementById('box');

получает элемент со страницы по его id.

const btns = document.getElementsByTagName('button');

получает псевдомассив (коллекцию) с элементами по имени тега.

После с помощью индекса можно получить определённый элемент, это можно сделать двумя способами:

- если в переменную нужно сразу ПОЛУЧАТЬ конкретный элемент:

const btns = document.getElementsByTagName('button')[1];

в переменную btns помещается только элемент с индексом 1 (вторая кнопка).

- если нужно ИСПОЛЬЗОВАТЬ определённый элемент:

const btns = document.getElementsByTagName('button');

console.log(btns[1]);

в переменную btns помещается вся html коллекция кнопок, но далее в консоль выводится только элемент с индексом 1.

Даже если элемент один .getElementsByTagName() вернёт html коллекцию/псевдомассив в котором будет содержаться только один элемент. ВАЖНО! можно ошибочно взаимодействовать с этим псевдомассивом считая, что раз он содержит один элемент то можно без проблем изменять его css свойства(к примеру), но такие действия возможны ТОЛЬКО при обращении к элементу непосредственно, а не к массиву! Это делается следующим образом:

console.log(btns[0]);

const circles = document.getElementsByClassName('circle');

получает псевдомассив (коллекцию) с элементами по имени класса.

css селекторы задаются в скобках.

<button name="first">1</button>

const firstBtn = document.getElementsByName('first');

метод, позволяющий получить элементы по атрибуту name, возвращает псевдомассив NodeList (статический список), используется редко.

РАЗНИЦА между HTMLCollection и NodeList в том, что HTMLCollection – динамическая структура данных, а NodeList – статическая. HTMLCollection обновляется каждый раз, когда меняется количество элементов, полученных с помощью соответствующего метода. А NodeList не меняется после формирования, даже если меняется HTML код страницы.

1. Методы которые появились позже(более функциональные):

const hearts = document.querySelectorAll('.heart');

самый популярный метод, возвращает псевдомассив NodeList с элементами/ом. Во внутрь круглых скобок можно поместить ЛЮБОЙ css селектор(#id, .class, вложенность классов, атрибуты, комбинации всего вышеперечисленного и т.д.), поддерживает вложенность. Псеводмассив полученный с помощью этого метода имеет один метод – for each, позволяющий перебрать его и выполнить действия с каждым из его элементов.

hearts.forEach(item => {

    console.log(item);

});

const firstHeart = document.querySelector('.heart');

получает первый элемент на странице по выбранному селектору. Метод удобно использовать когда на странице есть элементы с уникальным css селектором.

ДЕЙСТВИЯ С ЭЛЕМЕНТАМИ НА СТРАНИЦЕ

const box = document.getElementById('box');

console.dir(box);

позволяет посмотреть на полученный элемент в качестве объекта. Среди множества свойств, есть style – это объект внутри объекта. Все свойства/стили, которые написаны в объекте style – это inline свойства/стили(не те которые получены из style.css, а те которые прописываются в атрибуте элемента).

<div class="box" id="box" style="background-color: blue; width: 500px;"></div>

Приоритет у inline стилей самый высокий – не важно какие стили заданы в style.css, inline стили их переопределят это сделано специально для простоты изменения стилей с помощью JS.

ВАЖНО, когда в JS прописываются стили, они должны быть прописаны точно так же как и в css:

box.style.width = '500px';

circles = document.getElementsByClassName('circle'),

circles.style.backgroundColor = 'purple';

такой код ничего не изменит, потому что в переменной circles находится ПСЕВДОМАССИВ (у которого нет вложенного объекта style), а НЕ ЭЛЕМЕНТ! Правильное обращение к элементу:

circles[0].style.backgroundColor = 'purple';

btn[1].style.borderRadius = '100%';

формат записи, где btn – переменная в которой находится псевдомассив с кнопками, [1] – второй элемент массива, style – стиль элемента, borderRadius – css свойство записанное в CamelCase, 100% - значение задаваемое этому свойству элемента.

Чтобы назначить сразу несколько инлайн стилей используется свойство .cssText, стили внутри этого свойства прописываются в нормальном css формате(не camelCase!):

box.style.cssText = 'background-color: red; width: 500px';

Метод позволяет быстро формировать inline стили, в том числе подставлять переменные используя бэктики и интерполяцию:

box.style.cssText = `background-color: red; width: ${num}px`;

полезно когда определённые параметры расчитываются динамически(например ширину модального окна в зависимости от устройства пользователя).

Если над несколькими элементами нужно произвести одни и те же действия, можно использовать цикл(почти не используется):

for (let i = 0; i < hearts.length; i++) {

    hearts[i].style.backgroundColor = 'blue';

}

перебирающую конструкцию for of или для псевдомассивов полученных с помощью .querySelectorAll, действия с элементами можно произвести с помощью метода for each:

heart.forEach(function(item, i, hearts) {

    item.style.backgroundColor = 'gold';

});

где item – элемент; i – индекс/номер элемента/итератор (задавать не обязательно); hearts – массив с элементами (задавать не обязательно). Запись с помощью стрелочной callback функции и без лишних параметров:

hearts.forEach(item => {

    item.style.backgroundColor = 'gold';

});

Основные методы для работы с элементами страницы:

Часто части сайта генерируются при помощи JS, на этом принципе построен React, где почти все элементы создаются при помощи скрипта. Для создания таких элементов на лету используется: document.createElement(‘div’) – создаёт элемент.

const div = document.createElement('div');

Созданный элемент существует только внутри JS, на странице он не появится.

const text = document.createTextNode('Text');

Создаёт текстовый узел(ноду=node=элемент на странице) без html тэга. Метод используется очень редко.

Для стилизации созданных в JS элементов страницы обычно используются заранее созданные в css классы, а не множество строк .style. … В реальных проектах чаще всего происходит работа с css классами, а не с inline стилями.

div.classList.add('black');

где div – имя переменной в которой хранится заранее созданный div; .classList – свойство со списком классов; .add – добавить(так же существует метод .toggle позволяющий переключать класс в зависимости от каких-либо действий); black – имя класса. Метод производит удаление, добавление, переключение, проверку на содержание и определение количества применённых классов к элементу.

Любой элемент можно вставить в конец, вставить после определённого элемента, удалить или заменить, но только по отношению к другому элементу(необходимо для чёткого обозначения места размещения элемента).

Современные методы для работы со страницей(не работают в устаревших браузерах):

Отличительная особенность – в большинстве этих методов идёт обращение напрямую к элементу, заранее получать родителя больше не нужно!

document.body.append(div);

добавляет в конец родителя заранее созданный элемент div с заранее определёнными стилями, хранящийся в переменной div. Body – имя тэга-родителя, родителем может быть любой заранее полученный элемент:

document.querySelector('.wrapper').append(div);

Элемент-родитель не обязательно помещать в переменную, если он больше нигде не будет использоваться. А если будет, то:

wrapper.append(div);

wrapper.prepend(div);

Метод добавляет элемент в начало родителя

hearts[0].before(div);

Метод .before вставляет элемент (div) перед указанным в начале элементом(hearts[0]).

hearts[0].after(div);

Метод .after вставляет (div) после указанного в начале элемента(hearts[0]).

circles[0].remove();

Метод .remove удаляет элемент(в данном случае с индексом 0 псевдомассива circles).

hearts[0].replaceWith(circles[0]);

Метод .replaceWith заменяет один элемент (herats[0]) другим (circles[0]).

Немного устаревшие, но встречающиеся методы:

Ранее все методы работали только через родителя!

document.body.appendChild(div);

добавляет в конец body заранее созданный элемент div с заранее определёнными стилями, хранящийся в переменной div.

wrapper.appendChild(div);

то же, но div вставляется в конце оболочки wrapper(переменная wrapper содержит родительский элемент – div с классом wrapper заранее полученный с помощью .querySelector или другого метода).

document.body.insertBefore(div, circle[1]);

вставляет заранее созданный и хранящийся в переменной элемент в начало body перед указанным элементом. Метод .insertBefore имеет 2 параметра – что вставлять и перед чем вставлять, если не указать второй, то будет работать как .appendChild().

document.body.removeChild(circle[2]);

удаляет элемент со страницы.

wrapper.replaceChild(circles[0], hearts[0]);

заменяет один заранее полученный элемент другим заранее полученным элементом. Первый указывается элемент на который нужно заменить(новый), вторым – элемент который нужно заменить(старый).

Добавление текста или HTML кода в элемент:

div.innerHTML = "<h1>Hello!</h1>";

создание html разметки/структуры или текста внутри заранее созданного/полученного элемента.

div.textContent = 'Hello!!!';

добавляет только текст, используется когда данные помещаемые в элемент принимаются от пользователя и нужно обезопасить страницу от возможного попадания вредоносного кода.

Как вставить кусочек html кода перед или после определённых тегов?

div.insertAdjacentHTML('beforebegin', '<h2>Hello!</h2>');

Метод применяется к созданному/полученному элементу и имеет 2 параметра – ключевое слово и внедряемый html код.

beforebegin – вставляет код перед элементом; afterbegin – вставляет код в начало элемента; beforeend – вставляет код перед концом элемента; afterend – вставляет код после элемента.

Чтобы получить конкретные элементы, не обязательно получать к ним доступ именно через document. Можно получить доступ к родителю и через родителя получить доступ ко вложенным элементам:

const wrapper = document.querySelector('.wrapper'),

    hearts = wrapper.querySelectorAll('.heart'),

    oneHeart = wrapper.querySelector('.heart');

Это удобно, т.к. в эти псевдомассивы попадут только элементы которые подойдут по определённому селектору и имеют определённого родителя. Область поиска формулируется более чётко.

СОБЫТИЯ И ИХ ОБРАБОТЧИКИ

Событие – сигнал от браузера о том, что что-то произошло(клик, двойной клик, наведение мыши, заполнение формы, отправка данных и т.д.)

Для работы с событием нужен обработчик. Обработчик события – это функция, которая срабатывает, когда событие уже произошло(нажал на меню – меню открылось, нажал на копку отправить – данные ушли на сервер).

Существует 3 способа назначить обработчик событий:

1. Используя атрибут вписать код прямо в html разметку. Почти не используется(используется не программистами) или только для маленького кода (подключения метрики-количество посещений сайта):

<button onclick="alert('Нажата кнопка 1')" id="btn">Нажми меня</button>

1. Использовать свойство дерева DOM для событий. П олучить элемент с помощью document. … и задать функцию обработчику событий этого элемента:

let btn = document.getElementsByTagName('button');

btn[0].onclick = function() {

    alert('Нажата кнопка 1');

};

Минусы этих методов – одному обработчику назначается одна функция, назначение других функций перезаписывает предыдущие. Обработчики, назначенный с помощью такого синтаксиса продолжают работать бесконечно и просто-так не удаляется(не удаляются после отработки).

1. Получить элемент(ы) и использовать метод .addEventListener (более современный способ):

btn[0].addEventListener('click', function() {

    alert('Нажата кнопка 1');

});

Можно назначать несколько функций/действий на одно событие:

const btn = document.querySelector('button');

btn.addEventListener('click', () => {

    alert('Click');

});

btn.addEventListener('click', () => {

    alert('Second click');

});

Оба действия будут выполнены одно за другим.

btn[0].addEventListener('click', function(event) {

    console.log('Произошло событие ' + event.type + ' на элементе ' +

event.target);

});

Используя объект event получаем информацию о событии и элементе, чтобы использовать её в дальнейшем. Event передаётся как аргумент в callback функцию, называть его можно как угодно, но обычно event или е, он всегда передаётся первым аргументом. Например, можно скрыть элемент:

let target = event.target;

target.style.display = 'none';

или удалить со страницы:

btn.addEventListener('click', (e) => {

    e.target.remove();

});

Метод .removeEventListener позволяет убрать/удалить обработчик событий, в качестве аргумента ему ОБЯЗАТЕЛЬНО должна быть передана та же callback функция, что была ранее передана методу .addEventListener. Выглядит так:

const deleteElement = (e) => {

    console.log(e.target);

};

btn.addEventListener('click', deleteElement);

btn.addEventListener('click', deleteElement);

Всплытие событий – это когда обработчик событий срабатывает сначала на самом вложенном элементе, затем на его родителе, если он есть, затем выше и выше(если назначен).

<div class="overlay">

        <button id="btn">Нажми меня</button>

    </div>

const btn = document.querySelector('button');

const overlay = document.querySelector('.overlay');

const consoleMsg = (e) => {

    console.log(e.target);

    console.log(e.type);

};

btn.addEventListener('click', consoleMsg);

overlay.addEventListener('click', consoleMsg);

Кнопка находится внутри блока overlay. Получаем оба этих элемента, назначаем им обработчик событий одно и ту же функцию. При клике по кнопке, обработчик начале срабатывает на ней, а потом на её родителе overlay’е, причем в event.target в обоих случаях содержится кнопка – элемент на котором произошло изначальное событие, в консоли будет:

<button id=​"btn">​Нажми меня​</button>​

click

<button id=​"btn">​Нажми меня​</button>​

click

НО! если выводить свойство e.currentTarget(используется редко), то в консоли будет:

<button id=​"btn">​Нажми меня​</button>​

click

<div class=​"overlay">​…​</div>​

click

Есть так же способ отменять вспылите событий, но используется редко.

Чтобы отменить стандартное поведение браузера есть 2 способа:  
1) Если обработчик назначен в формате on<событие>, вернуть false в функции обработчика событий, устаревший способ, почти не используется:

let link = document.querySelector('a');

link.onclick = function(event) {

    console.log(event.target);

    return false;

};

2) Метод .preventDefault() объекта события event. ВСЕГДА помещается в самое начало функции:

let link = document.querySelector('a');

link.addEventListener('click', function(event) {

    event.preventDefault();

console.log(event.target);

});

Отменяет стандартное поведение браузера относительно объекта event, например отменяет переход по ссылке. Часто используется в вэб приложениях.

Псевдомассиву назначить обработчик событий нельзя! Его необходимо перебрать и на каждый отдельный элемент навесить обработчик.

let btn = document.querySelectorAll('button')

btn.forEach(function(item) {

    item.addEventListener('mouseleave', function() {

        item.innerHTML = 'Вышли!';

    });

});

Самый предпочтительный метод для назначения обработчиков событий нескольким элементам!

Метод назначения обработчика событий для нескольких элементов(кнопок). Получаем псевдомассив кнопок с помощью .querrySelectorAll, назначаем каждому item(кнопке) обработчик

.addEventListener, который при выходе курсора за пределы кнопки меняет текст внутри неё.

Опции события

.addEventListener принимает 3 аргумента: событие, функцию обработчик и опции(например once – событие происходит единоразово и не нужно прописывать .removeEventListener). Третьим аргументом можно встретить и false(это необязательно) – это значит, что опции не назначены.

const btns = document.querySelectorAll('button');

btns.forEach(btn => {

    btn.addEventListener('click', consoleMsg, { once: true });

});

НАВИГАЦИЯ ПО DOM ЭЛЕМЕНТАМ, data АТРИБУТЫ, ПРЕИМУЩЕСТВО for\_of

console.log(document.documentElement);

выдаёт в консоль полное содержание тэга html

console.log(document.body.childNodes);

Свойство .childNodes позволяет получить все узлы, находящиеся внутри родителя; в примере выше, выдаёт в консоль nodelist – список узлов, которые являются детьми(вложены в) body.

1. *NodeList(6) [text, div.wrapper, text, comment, text, script]*
   1. 0: text
   2. 1: div.wrapper
   3. 2: text
   4. 3: comment
   5. 4: text
   6. 5: script
   7. 6: text
   8. 7: comment
   9. 8: text
   10. 9: script
   11. 10: text
   12. length: 11
   13. \_\_proto\_\_: NodeList

0. – текстовая нода, перенос строки после body; 1. – элемент div.wrapper; 2. – перенос строки; 3. – комментарий; 4. – перенос строки; 5. – подключение скриптового файла; 6. – 10. – динамические узлы, которые подключает live server в VSCode.

Разница между DOM элементами и DOM узлами – каждая сущность находящаяся на странице будет узлом, но не каждый узел будет элементом. Всё что в виде тэгов – это элементы, все что скорее всего невидимо(переносы строк, текстовые вставки) – это узлы/ноды.

console.log(document.body.firstChild);

console.log(document.body.lastChild);

Свойства .firstChild и .lastChild позволяют получить первый и последний узел/ноду/ребенка родителя соответственно.

console.log(document.querySelector('#current').parentNode);

Свойство .parentNode позволяет получить узел-родитель полученного ранее элемента.

console.log(document.querySelector('#current').parentNode.parentNode);

Двойное использование данного свойства позволяет получить родителя родителя элемента полученного вначале.

Дата атрибуты

При написании скриптов в html не хватает ориентиров(можно расставить id но у этого способа есть минусы и он не универсален) для этого были введены дата атрибуты.

<li data-current="3">3</li>

Синтаксис: первым идёт обязательное слово data затем произвольное слово, желательно говорящее о том для чего использован дата атрибут, к примеру data-close – закрытие чего-либо/data-modal – модальное окно. Атрибуту может быть присвоено значение, также он может быть просто прописан, тогда он приравнивается к true: <li data-current>3</li> эквивалентно

<li data-current=true>3</li>.

Получение элементов с дата атрибутами:

console.log(document.querySelector('[data-current="3"]'));

Получаются как и другие элементы с html атрибутами, синтаксис [атрибут=”значение”].

console.log(document.querySelector('[data-current="3"]').nextSibling);

console.log(document.querySelector('[data-current="3"]').previousSibling);

Свойства .nextSibling и .previousSibling позволяют получить следующего или предыдущего соседа/узел/ноду элемента, полученного в левой части выражения.

console.log(document.querySelector('[data-current="3"]').nextElementSibling);

А свойства .nextElementSibling и .previousElementSibling уже позволяют получить соседние элементы, а не ноды.

Существуют свойства .firstElementChild .lastElementChild позволяющие получать первый и последний элементы, а так же .parentElement – для получения родительского элемента. Но для .childNodes аналога с элементами нет, но его можно создать перебрав полученную псевдоколлецию с помощью for of, т.к. NodeList.forEach() поддерживается не во всех браузерах), а так же потому, что с помощью for of можно останавливать цикл, пропуская итерацию(continue) либо его полностью прерывать(break):

for (let node of document.body.childNodes) {

    if (node.nodeName == '#text') {

        continue;

    } else if (node.nodeName == '#comment') {

        continue;

    }

    console.log(node);

}

Результат работы цикла – в консоли будут только элементы (не переносы text и не комментарии comment).

СОБЫТИЯ НА МОБИЛЬНЫХ УСТРОЙСТВАХ

В мобильных браузерах нет кликов, но есть тапы – прикосновения пальцами, которых может быть несколько. Мобильные браузеры изначально нормально работают со всеми сайтами, если в десктопной версии есть событие клика, то в мобильной версии он так же сработает от тапа, т.к. JS запускает сразу серию событий, чтобы ничего не пропустить.

Существует 6 событий работающих в мобильном браузере:

touchstart – аналог клика, возникает при касании к элементу.

touchmove – перемещение пальца после прикосновения к элементу.

touchend – палец перестал соприкасаться с сенсором.

touchenter – палец зашел на элемент на странице.

touchleave – палец покинул вышел за границы элемента.

touchcancel – палец вышел за пределы браузера, событие перестало существовать.

При навешивании обработчика события на мобильном устройстве рекомендуется сразу в параметрах callback функции передавать объект события event и с помощью него отменять стандартное поведение браузера методом .preventDefault().

box.addEventListener('touchstart', (e) => {

        e.preventDefault();

    });

Существует 3 главных свойства при работе с сенсорными устройствами:

event.touches

объект содержащий список (TouchList) всех прикосновений к сенсору в данный момент.

event.changedTouches

аналогично даёт список пальцев, которые участвуют в текущем событии(прим. если событие touchend то список будет содержать и палец который был убран, даже если оставшиеся все ещё на экране).

event.targetTouches

получаемый объект содержит только те касания, которые взаимодействовали с определённым элементом.

Комбинируя разные виды событий можно получать те, которые явно не прописаны, например свайп или щипок/разглаживание двумя пальцами(уменьшение/ увеличение чего-либо).

В реальных проектах большие и сложные события обычно не создаются с нуля а используются готовые лёгкие библиотеки типа https://hammerjs.github.io

РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

Нужны для взаимодействия со строками, удалять, заменять части слов, ограничивать ввод определённых знаков и т.д. РВ всегда состоят из двух частей – паттерны и флаги. Использование РВ сокращает количество кода!

Способы записи:

1. В виде конструктора:

new RegExp('pattern', 'flags');

1. Используя 2 косых слеша:

/pattern/flags

Pattern или шаблон – это то, что необходимо найти: буквы, цифры, знаки и т.д.

Flags – вспомогательные символы:

i – флаг регистра (если нужно найти что-то вне зависимости от регистра)

g – флаг глобальности (ищем не только первое вхождения/появление, но и все остальные)

m – флаг многострочности

let ans = prompt('Введите ваше имя');

let reg = /n/i;

console.log(ans.search(reg));

Если введено имя содержащее n, метод .search(ищет всегда только первое совпадение, не работает с флагом g) возвращает позицию этой буквы в введённом имени, если буквы нет, вернёт -1.

let reg = /n/;

console.log(ans.match(reg));

.match с флагом глобальности g даёт массив со всеми совпадениями, которые были найдены в указанной строке.

let pass = prompt('Введите пароль');

console.log(pass.replace(/./g, "\*"));

.replace заменяет символы . – (точка) означает, что будет выполнен поиск или замена любых(первых попавшихся) символов, которые попадут в строку /g, “\*” – глобально, т.к. все символы будут заменены на \*

alert('12-54-56'.replace(/-/g, ":"));

все – заменить на :

console.log(reg.test(ans));

Проверяет работу регулярного выражения в строке ans. Если находит указанную букву – true, если нет – false.

Классы символов:

\d – digits цифры \D – НЕ число

\w – words слова(буквы) \W – НЕ слово(буква)

\s – spaces пробелы \S – НЕ пробел

let ans = prompt('Введите число');

let reg = /\d/g;

console.log(ans.match(reg));

даёт массив со всеми найденными цифрами.

let str = 'My name is R2D2';

console.log(str.match(/\w\d\w\d/i));

Пример комбинирования классов символов для поиска “R2D2” в строке без учёта регистра. На консоли массив со всей информацией о нахождении этого выражения.

let str = 'My name is / R2D2';

console.log(str.match(/\//i));

Способ поиска спецсимволов, таких как $, \* и т.д.

В примере ищет слэш / ,для этого в паттерне ставится обратный ЭКРАНИРУЮЩИЙ слэш \.

ASYNC, DEFER, ДИНАМИЧЕСКИЕ СКРИПТЫ

<script defer src="script.js"></script>

defer – html атрибут тэга script, который сообщает браузеру, что ему нужно продолжать обрабатывать страницу(строить DOM) и загружать указанный скрипт в фоновом режиме, а затем запустить скрипт когда он загрузится. Особенности:

1) Скрипты с defer никогда не блокируют загрузку страницы (скорее всего независимо от расположения тэга <script> (в head/начале body/конце body) страница продолжает загружаться).

2) Скрипты с атрибутом defer выполняются когда DOM дерево уже готово. Но срабатывают ещё до события DOMContentLoaded, что не особо важно на практике.

Скрипты с атрибутом defer загружаются последовательно не зависимо от размере, по мере того, как браузер встречает их в html разметке.

<script async src="script.js"></script>

Async – html атрибут тэга script. Особенности:

1) Страница не ждёт выполнения/загрузки асинхронных скриптов, содержимое просто обрабатывается и отображается.

2) События DOMContetLoaded и асинхронные скрипты с тэгом async не ждут друг друга.

3) Если стоит атрибут async, скрипт начинает загружаться в фоновом режиме как только до него доходит страница, но при этом запускается как только он был загружен(по готовности) не ожидая других событий. Остальные скрипты не ждут async так же как скрипты с async не ждут остальные скрипты.

Атрибут применяется для скриптов с метриками и счётчиками, которые не вовлечены в DOM структуру. Т.е. перед применением async нужно точно знать, что этот скрипт не зависит от DOM структуры или других скриптов

Динамически загружаемый скрипт:

const script = document.createElement('script');

script.src = 'js/test.js';

document.body.append(script);

Способ добавления скрипта на старницу. В первой строке создаётся переменная содержащая элемент с тэгом <script>. Во второй строке свойству src переменной script присваивается значение соответствующее месту расположения скрипта. В третьей строке с помощью метода .append() созданный элемент помещается в конец <body>. Такой скрипт начнёт загружаться только после того, как будет добавлен в документ, т.е. после выполнения третьей строчки.

Динамически загружаемые скрипты по умолчанию ведут себя как async и не будут ничего ждать. Чтобы отменить это поведение или изменить относительный порядок скриптов при их загрузке необходимо установить атрибут async динамически загружаемого скрипта в значение faslse:

script.async = false;

В таком случае он будет вести себя как обычный скрипт и загружаться последовательно относительно других скриптов.

Пример Youtube iframe API инструмент для добавления роликов с ютуба на страницу, использует такой способ динамического формирования скриптов.

classList И ДЕЛЕГИРОВАНИЕ СОБЫТИЙ

Методы свойства classList:

<button class="blue some"></button>

const btns = document.querySelectorAll('button');

console.log(btns[0].classList.item(0));

Метод .item() позволяет получить класс элемента, который располагается под определённым индексом, в данном примере класс находящийся под нулевым индексом у элемента <button> это blue.

console.log(btns[0].classList.add('red'));

Метод .add() добавляет указанный класс элементу, в данном примере класс red заранее прописан в style.css и после его добавления цвет кнопки изменится на красный.

console.log(btns[0].classList.remove('blue'));

Метод .remove() удаляет указанный класс у элемента, в данном примере удаляется класс blue (он не нужен так как его css свойства были переназначены после добавления класса red выше).

console.log(btns[0].classList.toggle('blue'));

Метод .toggle() позволяет «тоглить»/переключать классы, если у элемента нет такого класса, он будет добавлен, если есть, то убран. Хоть класс и blue помещается в конец списка классов, но цвет кнопки может остаться прежним если в style.css присутствуют другие стили переопределяющие blue или описанные ниже его, в данном случае red описан ниже.

if (btns[1].classList.contains('red')) {

    console.log('red');

}

Метод .contains() проверяет наличие класса у элемента и возвращает true или false в зависимости от результат. Часто используется в условиях для выполнения каких-либо действий, этот механизм позволяет динамически преобразовывать страницу.

Пример: грамбургер меню, при клике на которое добавляется класс активности гамбургеру и меню, после следующего клика выполнится проверка условия, если класс активности уже присутствует, то меню и гамбургер закроются, классы активности удалятся. Ещё пример:

btns[0].addEventListener('click', () => {

    if (!btns[1].classList.contains('red')) {

        btns[1].classList.add('red');

    } else {

        btns[1].classList.remove('red');

    }

});

Первой кнопке назначается обработчик событий по клику. Если при клике на первую кнопку, вторая кнопка НЕ содержит класс red, он добавляется, иначе(если содержит) он удаляется.

То же поведение но с .toggle():

btns[0].addEventListener('click', () => {

    btns[1].classList.toggle('red');

});

Но в сложных скриптах иногда нужно вручную проверять есть ли какой-то класс.

Для того чтобы classList работал с несколькими классами не нужно дублировать команду, достаточно перечислить необходимые классы в методе через запятую:

console.log(btns[0].classList.add('red', 'yellow', 'purple'));

className

Существует устаревшее свойство содержащее все классы элемента в виде одной строки:

console.log(btns[0].className);

Считается устаревшим, т.к. для манипуляций с классами приходится изменять эту строку какими либо методом.

Делегирование событий

Делегирование решает следующую проблему - можно навесить обработчик на уже существующие элементы, но на динамически генерируемых элементах(добавляемых в процессе) его не будет, т.к. они не обрабатываются циклом в котором назначается обработчик.

Выход – работать с родителем, т.е. назначать обработчик родителю, а внутри уже проверять с чем происходит взаимодействие, а именно назначать функции потомкам если они подходят под определённые параметры.

Используется когда есть множество обработчиков, экономит код и память браузера.

let btnBlock = document.querySelector('.btn-block'),

    btns = document.getElementsByTagName('button');

btnBlock.addEventListener('click', function(event) {

    if (event.target && event.target.tagName == 'BUTTON') {

        console.log('Button');

    }

});

Пример делегирования событий: получаем обёртку кнопок и сами кнопки, элементу-обёртке с кнопками btnBlock назначается обработчик событий по клику. Проверяется условие Если цель клика это кнопка и имя тэга элемента BUTTON (особенность: tagName ВСЕГДА! Пишется капсом), в консоль выводится соответствующее сообщение. event.target прописан в условии потому что в html не все элементы поддерживают событие клика(например на <br> кликнуть нельзя), фактически это проверка на существование элемента и возможности кликнуть по нему как таковой. Проверка event.target является рекомендацией Google! Клики по элементу родителю ничего не выводят, потому что его имя тэга <div>.

В делегировании если элемент подходит под условия, то на нём будет срабатывать код внутри условия (написанная или переданная функция). Действия с родителем делегируются на его потомков. Можно добавлять любое количество потомков, код будет работать. А при навешивания обработчика методом перебора с помощью .forEach() код не сработает не зависимо от того где в коде описано создание новых элементов (перед или после .forEach()).

В условии можно проверять не только .tagName, но и .nodeName и .classList. Так же используются:

event.target.classList.contains('first')

метод .contains() проверяет есть у элементов-целей события класс ‘first’

event.target.matches('button.first')

метод .matches() ищет среди элементов-целей совпадения – элемент button с классом .first. Любим сотрудниками Google.

СОЗАДЁМ ТАБЫ В НОВОМ ПРОЕКТЕ (Практика 7)

(работа с лэндингом фитнес/food)

Табы – это вкладки на сайте. Алгоритм создания табов:

1. Создать функцию которая будет скрывать ненужные табы.
2. Показать нужный таб.
3. Назначить обработчики событий на меню, чтобы манипулировать вышеописанными функциями.

Скрипты обычно начинаются с получения переменных с которыми будет производиться дальнейшее взаимодействие:

window.addEventListener('DOMContentLoaded', () => {

    const tabs = document.querySelectorAll('.tabheader\_\_item'),

        tabsContent = document.querySelectorAll('.tabcontent'),

        tabsParent = document.querySelector('.tabheader\_\_items');

});

В вёрстке присутствует класс активности .tabheader\_\_item\_active который необходимо будет удалять и добавлять к соответствующим пунктам меню по которым кликнули.

Решение задачи № 1:

function hideTabContent() {

tabsContent.forEach(item => {

            item.style.display = 'none';

        });

        tabs.forEach(item => {

            item.classList.remove('tabheader\_\_item\_active');

        });

    }

Вначале с помощью перебора методом .forEach всех элементов с классом .tabcontent находящихся внутри псевдомассива внутри переменной tabsContent устанавливаем инлайн свойство display в значение none чтобы скрыть/не отображать эти элементы на странице. Затем перебираем псевдомассив элементов с классом .tabheader\_\_item (пункты меню) находящийся в переменной tabs и удаляем у этих элементов класс активности.

Решение задачи № 2:

function showTabContent(i = 0) {

        tabsContent[i].style.display = 'block';

        tabs[i].classList.add('tabheader\_\_item\_active');

    }

Функция showTabContent в качестве аргумента принимает номер таба, который будет отображаться – i, по умолчанию первый таб (i = 0, синтаксис из ES6). Для отображения контента i-того таба, в инлайн стилях значение свойства display изменяется на block. А класс активности .tabheader\_\_item\_active добавляется i-тому пункту меню в противоположность предыдущей функции.

После написания первых двух функций, они вызываются для создания изначального состояния страницы:

hideTabContent();

showTabContent();

Решение задачи № 3:

Черновая логика работы – после клика по определённому пункту меню(.tabheader\_\_item), необходимо определить его номер = индекс в псевдомассиве в переменной tabs и вызвать функцию showTabContent с этим номером в качестве аргумента, чтобы таб отобразился на странице.

Будем использовать делегирование событий для назначения обработчика клика родителю.

tabsParent.addEventListener('click', (event) => {

        const target = event.target;

        if (target && target.classList.contains('tabheader\_\_item')) {

            tabs.forEach((item, i) => {

                if (target == item) {

hideTabContent();

showTabContent(i);

                }

            });

        }

    });

});

Если в скрипте часто используется конструкция event.target её лучше вынести в отдельную локальную переменную(target), считается распространённой практикой. Затем проверяется условие – если элемент по которому кликнули «кликабельный» и среди списка его классов есть .tabheader\_\_item (т.е. проверка является ли он пунктом меню), то запускается перебор псевдомассива в переменной tabs. Если тот элемент по которому произошел клик(target) является тем, который сейчас перебирается(item) target == item, будут вызваны две функции: hideTabContent скроет все табы, а showTabContent(i) отобразит таб с индексом i, где i – это индекс элемента в псведомассиве, который в данном условии совпал.

В реальных проектах не всегда используются инлайн стили, а так же не всем нравится резкое переключение между табами без переходов/анимаций. Для этого нужно прописать дополнительные классы и правила в style.css:

.show {

    display: block

}

.hide {

    display: none

}

.fade {

    animation-name: fade;

    animation-duration: 1.5s;

}

/\* Правило @keyframes устанавливает ключевые кадры при анимации элемента от...до... \*/

@keyframes fade {

    from {

        opacity: 0.1;

    }

    to {

        opacity: 1;

    }

}

Код переписывается с целью задействовать данные классы вместо инлайн стилей:

01: window.addEventListener('DOMContentLoaded', () => {

    02:     const tabs = document.querySelectorAll('.tabheader\_\_item'),

    03:         tabsContent = document.querySelectorAll('.tabcontent'),

    04:         tabsParent = document.querySelector('.tabheader\_\_items');

    05:     function hideTabContent() {

    06:         tabsContent.forEach(item => {

    07:             item.classList.add('hide');

    08:             item.classList.remove('show', 'fade');

    09:         });

    10:         tabs.forEach(item => {

    11:             item.classList.remove('tabheader\_\_item\_active');

    12:         });

    13:     }

    14:     function showTabContent(i = 0) {

    15:         tabsContent[i].classList.add('show', 'fade');

    16:         tabsContent[i].classList.remove('hide');

    17:         tabs[i].classList.add('tabheader\_\_item\_active');

    18:     }

    19:     hideTabContent();

    20:     showTabContent();

    21:     tabsParent.addEventListener('click', (event) => {

    22:         const target = event.target;

    23:         if (target && target.classList.contains('tabheader\_\_item')) {

    24:             tabs.forEach((item, i) => {

    25:                 if (target == item) {

    26:                     hideTabContent();

    27:                     showTabContent(i);

    28:                 }

    29:             });

    30:         }

    31:     });

    32: });

В строках 7-8 и 15-16 необходимо использовать комбинацию методов добавления и удаления классов, чтобы классы не переопределяли друг друга.

УПРАВЛЕНИЕ ВРЕМЕНЕМ ПРИМИНЕНИЯ СКРИПТОВ

039 Скрипты и время их выполнения. setTimeout и setInterval

const timerId = setTimeout(function(text) {

    console.log(text);

}, 2000, 'Hello');

setTimeout – один раз вызывает функцию/событие на странице через определённый промежуток времени. Первым аргументом принимает как объявление функции(возможно анонимной) так и готовую функцию(не вызов!). Вторым аргументом принимает отсрочку времени выполнения после загрузки страницы в миллисекундах(2000). Третьим аргументом может быть аргумент для функции внутри (‘Hello’). Третий аргумент задействуется нечасто, но их может быть «бесконечное количество».

При сохранении setTimeout и/или setInterval в переменную, в неё фактически записывается числовой идентификатор этого таймера (Вызов setTimeout возвращает ЧИСЛО!). Они могут работать и без присвоения переменной и их может быть очень много, но для дальнейшей остановки/сброса таких «таймеров» лучше передавать их в виде переменной или числового идентификатора(если они, к примеру, не присвоены переменным).

clearTimeout – отменяет выполнение таймера в обработчике timer.

clearInterval(timerId);

Метод setInterval выглядит точно так же как и setTimeout, принимает такие же аргументы и останавливается так же с помощью clearInterval, но событие повторяется через указанный интервал.

Чтобы остановить работу таймера, вызовом clearInterval вне функции в которой локально прописан этот таймер (т.к. clearInterval его «не видит»), используют глобальную переменную в которую в дальнейшем будет записан идентификатор этого таймера:

let timerId;

btn.addEventListener('click', () => {

    timerId = setInterval(logger, 2000, 'Hello');

});

clearInterval(timerId);

НО! В данном случае clearInterval всё равно не будет работать, потому что он идёт в «потоке синхронного когда». В момент когда он вызывается, в timerId всё ещё записан undefined. clearInterval сработает тогда когда для timerId отработает хотя-бы единожды. Контролировать количества срабатываний можно если ввести новую переменную итератор, увеличивать её с каждым срабатыванием timerId и с помощью условия сбрасывать таймер после необходимого количества срабатываний:

const btn = document.querySelector('.btn');

let timerId,

    i = 0;

btn.addEventListener('click', () => {

    timerId = setInterval(logger, 500);

});

function logger() {

    if (i === 0) {

        clearInterval(timerId);

    }

    console.log('text');

    i++;

}

Чем рекурсивный setTimeout лучше чем setInterval ? Когда таймер с интервалом начинает работу, он не учитывает как долго будет выполняться функция внутри него. Это значит, что если функция выполняется дольше чем установлена задержка, следующее исполнение/срабатывание функции может произой сразу же. Для решения проблемы используется рекурсивный вызов setTimeout (функция вызывает сама себя). В старом примере:

let timerId = setTimeout(function log() {

    console.log('Hello');

    setTimeout(log, 2000);

});

Внутри тела функции, функция setTimeout вызывает саму себя. Таким образом интервал остаётся постоянным, т.к. вначале выполняются действия внутри функции, а потом происходит задержка.

В новом примере:

let id = setTimeout(function log() {

    console.log('Hello');

    id = setTimeout(log, 500);

}, 500);

Код всегда будет ждать строго отведённое ему время. Сначала срабатывает первый setTimeout, а через пол секунды начинает работу функция log, которая только после полной отработки своего тела через пол секунды снова вызовет setTimeout с собой же в виде аргумента(рекурсия).

Задержка между выполнением функций переданных setTimeout/setInterval может быть равна нулю, но по умолчанию, чтобы не возникало багов устанавливается 4 миллисекунды – это стандарт, который соблюдается почти всеми браузерами.

Анимация в JS

С помощью setTimeout/setInterval прописываются простейшие анимации. Принцип – постепенно, через определённый промежуток времени меняем свойства элемента (css свойства).

Раньше анимация элементов реализовывалась за счёт JS, но с появлением CSS3, анимация задаётся с помощью классов, а JS добавляет/удаляет эти классы у элементов. Это намного удобнее.

ДАТЫ В JS

040 Работа с датами

Date – специфический объект. Работа с датами является обязательным навыком.

const now = new Date();

console.log(now);

Выдаст 2021-06-09T10:41:56.558Z Формат: год-месяц-числоТчасы:минуты:секунды: миллисекундыZ. Т – разделитель(время).

В переменную now с помощью конструктора Date сохраняется текущая дата и время, которые обычно берутся из системы. В Date() могут передаваться различные аргументы:

1. Дату в виде строки. Часто используется если на странице присутствует инпут с типом date

<input type=”date”>. Смысл – в переменную записывается не просто строка, а указанная дата:

const now = new Date('2021-06-09');

console.log(now);

Выдаст: 2021-06-09T00:00:00.000Z

1. «Прямой» аргумент. Если передать следующий аргумент:

const now = new Date(2021, 6, 9, 20);

console.log(now);

Выдаст: 2021-07-09T17:00:00.000Z Получается 07 месяц, потому что месяцы в дате считаются с 0. Дата так же учитывает часовые пояса, т.к. на ПК часовой пояс +03:00, когда задаётся 20 то по Гринвичу это будет 17.

1. Количество миллисекунд для преобразования в дату. Миллисекунды, потому что любая

дата хранящаяся в JS, хранится в миллисекундах, это значение называется timestemp. Для работы с датами необходимо преобразовать их в числовой формат. Во всех компьютерных системах время считается по Unix времени – это количество секунд или миллисекунд прошедших от “Unix эпохи” – 00:00 1 января 1970 года. Это связано с тем, что в разных странах дату записывают по разному. Подтверждение:

const now = new Date(0);

console.log(now);

Выдаст: 1970-01-01T00:00:00.000Z

Для получения даты до 01.01.1970 00:00 необходимо использовать отрицательное значение миллисекунд. Пример:

const now = new Date(-9999999999);

console.log(now);

Выдаст: 1969-09-07T06:13:20.001Z

Основные методы объекта Date

1. Методы получения компонентов даты:

const now = new Date();

console.log(now.getFullYear());

.getFullYear() выдаёт текущий год(2021). Год всегда должен передаваться в Date четырьмя цифрами.

const now = new Date();

console.log(now.getMonth());

.getMonth() выдаёт текущий месяц (5=июнь в программировании).

const now = new Date();

console.log(now.getDate());

.getDate() выдаёт текущее число (9 на момент написания). Месяц и число обозначаемые одним символом передаются в Date так же одним символом без нуля!

Соответственно существуют методы .getHours() .getMinutes() .getSeconds() .getMilliseconds() для получения часов, минут, секунд и миллисекунд.

const now = new Date();

console.log(now.getDay());

.getDay() выдаёт номер дня недели (3). Важная особенность, нумерация начинается с воскресенья (0 нулевой день).

Все вышеперечисленные методы возвращают данные в соответствии с местным часовым поясом. Когда необходимо использовать часовой пояс UTC +00:00, у всех этих методов есть аналоги с UTC, пример:

const now = new Date();

console.log(now.getHours());

console.log(now.getUTCHours());

.getHours() выдаст 14, а .getUTCHours() выдаст 11 соответственно.

const now = new Date();

console.log(now.getTimezoneOffset());

.getTimezoneOffset() выдаёт разницу между текущим часовым поясом и UTC в минутах (-180).

const now = new Date();

console.log(now.getTime());

.getTime() выдаёт timestamp – количество миллисекунд прошедших с 01.01.1970 00:00 (1623239701838).

1. Методы установки даты:

Методы те же, только с приставкой set, кроме .getTimezoneOffset().

const now = new Date();

now.setHours(18);

console.log(now);

Выдаст в консоль VS Code 2021-06-09T15:14:34.330Z, особенность консоли разработчика внутри VS Code, которая ориентируется на UTC зону, а не на локальное время ПК. При запуске Live server в консоли будет Wed Jun 09 2021 18:14:41 GMT+0300 (Восточная Европа, летнее время) отработано правильно. Т.е. браузер ориентируется на локальную дату.

const now = new Date();

now.setHours(18, 40);

console.log(now);

Можно передавать значения следующих аргументов, которые идут за тем, который устанавливается. Пример выше, задаём 18 часов 40 минут, в консоли браузера будет Wed Jun 09 2021 18:40:45 GMT+0300 (Восточная Европа, летнее время). Синтаксис используется но является не очевидным.

В JS присутствует автоисправление даты. Пример:

const now = new Date();

now.setHours(40);

console.log(now);

В браузере будет Thu Jun 10 2021 16:24:22 GMT+0300 (Восточная Европа, летнее время), т.е. следующий день после текущего (на момент написания 09.06.21 15:24). Если скрипт устанавливает значение даты вне обычного диапазона значений, то объект пытается использовать автоисправление.

Альтернативный синтаксис создания даты с помощью метода .parse():

const now = new Date('2021-06-09');

new Date.parse('2021-06-09');

Во втором случае используется метод, чтобы трансформировать «спарсить» строку в объект Date, а в первом случае это происходит автоматически.

Даты можно использовать для измерения промежутков времени, для этого используются миллисекунды. Timestamp’ы/миллисекунды можно использовать напрямую, а можно использовать динамическое преобразование типов данных поставив унарный плюс перед объектом Date.

let start = new Date();

for (let i = 0; i < 100000; i++) {

    let some = i \*\* 3;

}

let end = new Date();

alert(`Цикл отработал за ${end - start} миллисекунд`);

В start хранится время перед началом выполнения цикла for. \*\* - возводит в указанную степень, оператор появился в ES7. В end хранится время после отработки цикла for. Не смотря на то, что Date объект, при использовании его для математических операций (подсчёте end – start) он ведёт себя как обычные цифры/количество миллисекунд. Если усложнить данную операцию то можно использовать в качестве бэнчмарка для оценки быстродействия системы.

<input type="date" id="start-date">

let dateStart = document.querySelector('#start-date').value;

dateStart = Date.parse(dateStart);

Вводимая в инпут человекочитаемая дата с помощью метода .parse преобразовывается в количество миллисекунд.

24\*60\*60\*1000

Количество милисекунд в сутках: 24 часа \* 60минут в часе \* 60 секунд в минуте \* 1000 миллисекунд в секунде.

.toISOString

Возвращает дату в формате ISO 8601 год-месяц-день время часовой пояс.

.substr(0, 10)

Позволяет «вырезать»(извлечь подстроку) из такой даты. Результат: год-месяц-день.

Объект Date мощный, но его форматирование довольно затруднительно(вызывает желание его «доработать»), поэтому для работы с датами обычно используются либо сампописные библиотеки либо готовые библиотеки типа moment.js

document.querySelector('button').onclick = function() {

    let dateStart = document.querySelector('#start-date').value;

    let dateEnd = document.querySelector('#end-date').value;

    dateStart = Date.parse(dateStart);

    dateEnd = Date.parse(dateEnd);

    let out = document.querySelector('.out');

    for (let i = dateStart; i < dateEnd; i = i + 24 \* 60 \* 60 \* 1000) {

        out.innerHTML += (new Date(i).toISOString().substr(0, 10)) + '<br>';

    }

};

Пользователь вводит начальную и конечную дату в инпуты с id #start-date и #end-date. Значения дат преобразовываются в миллисекунды методом .parse. После цикл for выводит в div с классом .out даты от начальной, заданной пользователем, до конечной не включая её (чтобы С ней, нужно условие i<= dateEnd) метод .tiISOString() преобразовывает формат даты из миллисекунд в дату формата ISO, .substring оставляет от неё год-месяц-день, <br> - каждая дата с новой строки.

041 Создаем таймер обратного отсчета на сайте (Практика 8)

Алгоритм действий:

1. Написать функцию, которая будет устанавливать таймер, т.е. получить элементы

отображающие на сайте дни/часы/минуты/секунды и запустить обратный отсчёт.

1. Реализовать функционал определяющий разницу между временем/ дедлайн/ сколько

осталось до конца акции (будем устанавливать в формате даты). Подразумевает вычисление времени у пользователя и вычисление разницы с таймером на сайте.

1. Написать функцию отвечающую за визуальное обновление таймера на сайте.

Таймеры могут работать по нескольким принципам: честные таймеры – чётко отсчитывают время до заданного, таймеры для продажи – запускаются при входе на страницу запоминая пользователя и ориентируясь только на конкретного пользователя отсчитывают свой промежуток времени.

Дата будет в формате строки типа '2020-6-11' (месяц и число лучше обозначить одним символом иначе баги при расчётах) потому что в будущем данный скрипт может быть подвязан к административной панели, где может быть инпут с типом дата, который будет возвращать подобную строчку с которой в последствии нужно работать. С такой строкой математические операции невозможны, для этого её вначале нужно преобразовать.

Решение задачи №2:

44:     const deadline = '2020-6-11'; //Дата окончания акции

45:

46:     function getTimeRemaining(endtime) {

47:         const t = Date.parse(endtime) - Date.parse(new Date()),

48:             days = Math.floor(t / (24 \* 60 \* 60 \* 1000)),

49:             hours = Math.floor((t / (60 \* 60 \* 1000)) % 24),

50:             minutes = Math.floor((t / 60 / 1000) % 60),

51:             seconds = Math.floor((t / 1000) % 60);

52:

53:         return {

54:             'total': t,

55:             'days': days,

56:             'hours': hours,

57:             'minutes': minutes,

58:             'seconds': seconds

59:         };

60:     }

В строке 44 создаётся переменная которая будет хранить в себе дату окончания акции в виде строки. Строка 46, объявляется функция, в качестве аргумента которой будет передаваться дата окончания акции. Задача функции вычислять разницу между датой окончания и текущей датой. В строке создаётся переменная t, которая будет хранить в себе разницу между датами в миллисекундах. Разница вычисляется с применением метода .parse() который преобразует строку в миллисекунды. Вторая часть выражения, для более понятного синтаксиса, выглядит как Date.parse(new Date()), но может быть сокращена до new Date(). Хранящееся в t значение необходимо превратить в оставшееся количество дней/часов/минут/секунд, эти значения вычисляются в строках 48-51. Значения округляются до целых с помощью Math.floor().

Часы/минуты/секунды – получаем как остаток от деления (оператор % даёт остаток от деления) общего количества миллисекунд в t на количество миллисекунд в часе/минуте/секунде и следом на стандартизированное значение этих единиц (чтобы не было 50 часов и подобного, 50/24 =2дня - отбрасываем и 2 часа – сохраняем в hours). Для того чтобы вернуть переменные из функции, в строке 53 возвращается объект, такой приём часто используется. В свойства объекта записываются переменные.

Решение задачи №1 и 3:

62:     function getZero(num) {

    63:         if (num >= 0 && num < 10) {

    64:             return `0${num}`;

    65:         } else {

    66:             return num;

    67:         }

    68:     }

    69:

    70:     function setClock(selector, endtime) {

    71:         const timer = document.querySelector(selector),

    72:             days = document.querySelector('#days'),

    73:             hours = document.querySelector('#hours'),

    74:             minutes = document.querySelector('#minutes'),

    75:             seconds = document.querySelector('#seconds'),

    76:             timeInterval = setInterval(updateClock, 1000);

    77:

    78:         updateClock();

    79:

    80:         function updateClock() {

    81:             const t = getTimeRemaining(endtime);

    82:

    83:             days.innerHTML = getZero(t.days);

    84:             hours.innerHTML = getZero(t.hours);

    85:             minutes.innerHTML = getZero(t.minutes);

    86:             seconds.innerHTML = getZero(t.seconds);

    87:

    88:             if (t.total <= 0) {

    89:                 clearInterval(timeInterval);

    90:             }

    91:         }

    92:     }

    93:

    94:     setClock(".timer", deadline);

В строке 70 пишем функцию setClock, которая будет устанавливать таймер на странице. В качестве параметров функции указываются selector – css селектор блока(родитель), в котором находятся элементы содержащие составные части таймера и endtime – время окончания акции. В строках 71-75! создаются переменные в которые помещаются элементы со страницы. Прямо внутри функции setClock, в строке 80 объявляется функция updateClock. Эта функция будет выполнять 3 основных действия. 1) В строке 81 рассчитывать время оставшееся на данный момент, т.е. в переменную t будет помещаться объект получаемый в результате работы функции getTimeRemaining. Далее свойства этого объекта будут помещены на страницу. 2) В строках 83-86 расчётные величины помещаются на страницу. 3) Запускать функцию updateClock каждую секунду и остановить таймер, когда время истекло. Для этого в строке 76! объявляется переменная timeInterval в которую помещается запуск setInterval с функцией updateClock через 1000 миллисекунд. В строках 88-90 прописано условие остановки таймера – если t.total/оставшееся количество миллисекунд <= 0, вызывается clearInterval с переменной в которой находится setInterval и таймер останавливается.

Т.к. все основные функции готов, в строе 94 происходит запуск таймера вызовом функции setClock с аргументами в виде селектора .timer (класс блока в котором находится таймер) и переменной deadline в которой находится дата окончания акции в виде строки(которая в перспективе может приходить из многих источников).

Необходимо скорректировать баги и недочёты: 1) подставлять 0 для первых двух значений, когда они выражены одним числом; 2) если обновлять страницу, происходит «мигание» таймера, потому что сначала подставляются значения из вёрстки и только через секунду они меняются на расчётные значения.

Решение проблемы №2:

Для того чтобы избежать мигания вёрстки, необходимо вызвать функцию updateClock в самом начале, стока 78. Алгоритм следующий: запускается функция setClock, создаёт все переменные, запускает setInterval, который впервые сработает только через секунду, чтобы не ждать это время, вызываем updateClock вручную.

Решение проблемы №1:

В строках 62-68 пишем функцию getZero с параметром в виде числа, которое она будет проверять. Если число >= 0 (исключается вероятность попадания отрицательного) и < 10 (работа только с односимвольными), то с помощью интерполяции функция будет возвращать комбинацию из нуля и этого числа(это уже будет тип данных строка, но в данном случае это неважно т.к. это значение будет просто помещено на страницу). Иначе функция просто вернёт это число. Чтобы это сработало, вызываем функцию в строках 83-86, передавая ей в качестве аргументов свойства объекта соответствующие элементам таймера - дни/часы/минуты и т.д. Результаты вызова помещаются в html элементов на странице.

ПАРАМЕТРЫ ДОКУМЕНТА, ОКНА И РАБОТА С НИМИ (Метрики)

Основные понятия в работе JS относительно человека:

- document – объект содержащий всю html структуру;

- window – окно, в котором показывается документ;

- screen – экран, весь видимый монитор, разные размеры монитора = разный screen. Используется редко.

Метрики относящиеся к элементам

Элементы на странице имеют много различных параметров, но чаще всего используется лишь 4-5 из них. Все они измеряются в пикселях px, но в коде они пишутся без единиц измерения.

Нижеперечисленные свойства доступны только для чтения, изменить их из JS нельзя

let width = box.clientWidth,

    height = box.clientHeight;

переменные width и height будут содержать свойства элемента box, а именно ширину контента в коробке, включая сам контект, падинги, но без полос прокруты и бордеров(границ).

let width = box.offsetWidth,

    height = box.offsetHeight;

Свойства .offsetWidth и .offsetHeight дают такие же параметры, но уже включающие размеры бордера(границ) и полосы прокрутки. НО! даются размеры только видимой части.

let width = box.scrollWidth,

    height = box.scrollHeight;

Эти свойства включают размеры всего контента, но не содержат ширину полосы прокрутки(15px). Для ширины это не страшно, т.к. скролы с горизонтальной прокруткой почти не используются.

Зачем получать эти свойства ? Т.к. в css могут использоваться разные относительные параметры, например view height, view width, %, которые ориентируются на размер монитора пользователя, нельзя/сложно получить определённый размер элемента в пикселях прямо из стилей. Поэтому ориентируясь на полученные значения ширины/высоты модального окна, можно к примеру увеличить его высоту и поместить туда нужный контент.

btn.addEventListener('click', function() {

    box.style.height = box.scrollHeight + 'px';

});

Обработчик события по клику на кнопку btn вызывает callback функцию, которая меняет высоту элемента на «прокручиваемую высоту», тем самым показывая весь текст внутри скрола.

Изменяемые свойства чаще всего применяемые на практике:

.scrollTop .scrollLeft – содержат ширину и высоту невидимой, уже прокрученной части элемента сверху или слева. Пример: используя .scrollTop можно добавить прогресс сколько % страницы уже было пролистано.

Если в css есть правило box-sizing: border-box(значит что width и height задают не параметры контента а всего блока и включают в себя значение полей(border) и границ(padding), т.е. блок за счёт этого становится меньше) это скажется на параметрах .clientWidth .clientHeight – они будут меньше чем без него, так же из значения вычитается размер полосы прокрутки 15рх. В реальных проектах box-sizing: border-box чаще всего присутствует.

Координаты в JS:

В CSS все расстояния отсчитываются от границы браузера(родителя) до границ элемента(левая – до левой, нижняя до нижней и т.д.)

В JS right отсчитывается от левой границы браузера до правой границы элемента, bottom - отсчитывается от верхней границы браузера до нижней границы элемента, left и top аналогично CSS.

Для JS – точка начала координат левый верхний угол.

box.getBoundingClientRect()

метод .getBoundingClientRect() возвращает объект, содержащий bottom, top, left, right, width, x, y. Все координаты, которые есть у элемента. Их можно получить отдельно, обратившись к свойствам этого объекта.

Как получить стили, которые уже были применены к элементу в css:

Это важно когда, к примеру, перед работой с элементом в JS важно знать показан он на странице или нет(параметр display). Для этого используются Computed стили – стили, которые уже были применены к элементу. Напрямую менять их нельзя, но можно получить и использовать в проверках каких-либо условий. За это отвечает метод .getComputedStyle() объекта window. Синтаксис:

const style = window.getComputedStyle(box);

В переменную style помещается объект с Computed стилями элемента box.

console.log(style.display);

Выведет значение параметра display элемента box.

Важные особенности метода .getComputedStyle():

1)С его помощью можно получать стили псевдоэлементов. С помощью JS с ними нельзя работать, т.к. нельзя получить их со страницы, это прописано в стандарте.

Для получения таких стилей нужно через запятую указать в методе второй аргумент – псевдоэлемент, который относится к указанному ранее элементу.

2)Инлайн стили (прописанные или добавляемые прямо в вёрстку) более приоритетны чем Computed стили (прописанные в css) и переопределяют их.

Метрики относящиеся к document и window

document.documentElement.clientWidth

document.documentElement.clientHeight

возвращает ширину/высоту страницы, не включает окно с url адресом, окно с консолью и т.д. А так же меняется в зависимости от их размеров.

console.log(document.documentElement.scrollTop);

возвращает значение насколько страница проскролена сверху. Часто применяется в скриптах когда страницу нужно переместить вверх или узнать насколько страница «отмотана». document.documentElement.scrollTop = 0; вернёт страницу в начало после скрола.

Методы перемещения по странице, относятся скорее к window:

scrollBy(x, y); - перемещение на х по горизонтали на у по вертикали относительно текущего местоположения страницы.

scrollTo(x, y); - перемещает в указанное место на странице, отсчитывается от начала.

043 СОЗДАЁМ МОДАЛЬНОЕ ОКНО (Практика 9)

Задача на урок: прописать функционал модального окна и назначить его вызов сразу на несколько триггеров.

Для того, чтобы пометить элементы-триггеры (у них могут быть разные классы и тэги), им назначаются data атрибуты:

<button data-modal class="btn btn\_dark">Связаться с нами</button>

С помощью подобного селектора, можно получить только те кнопки, которые отвечают за вызов модального окна.

 <div data-close class="modal\_\_close">&times;</div>

Аналогично с элементами закрывающими модальное окно.

Алгоритм построения скрипта:

1. Написать функцию отвечающую за открытие модальных окон на нескольких триггерах.
2. Написать функцию отвечающую за закрытие модальных окон.

Функции можно написать как с использованием . classList. add(‘show’) . classList. remove(‘hide’), так и с применением . classList. toggle(‘show’).

Классы show и hide имеют следующие стилистические правила:

.show {

    display: block

}

.hide {

    display: none

}

Вариант с .add()/.remove() :

098:     const modalTrigger = document.querySelectorAll('[data-modal]'),

099:         modal = document.querySelector('.modal'),

100:         modalCloseBtn = document.querySelector('[data-close]');

101:

102:     modalTrigger.forEach(item => {

103:         item.addEventListener('click', () => {

104:             modal.classList.add('show');

105:             modal.classList.remove('hide');

106:             document.body.style.overflow = 'hidden';

107:         });

108:     });

109:

110:     modalCloseBtn.addEventListener('click', () => {

111:         modal.classList.add('hide');

112:         modal.classList.remove('show');

113:         document.body.style.overflow = '';

114:     });

Строка 98-100 – получаем необходимые элементы со страницы. Строка 102-105 – на каждую кнопку навешиваем обработчик событий, который после клика по кнопке будет добавлять класс show и убирать класс hide у модального окна, чтобы оно отобразилось. Строка 106 – чтобы страница не вращалась под модальным окном, необходимо её зафиксировать(особенно когда модальное окно само внутри прокручивается), за отображение/прокрутку содержимого внутри блочного элемента отвечает свойство css overflow, значение hidden означает, что контент body будет обрезан без предоставления прокрутки, что и требуется. Строка 110-113 на кнопку закрытия модального окна/«крестик» назначается обработчик события клика, вызывающий callback функцию, которая добавит модальному окну класс hide и уберёт класс show тем самым скрыв его. В строке 113 установка свойству overflow значения '' даёт команду браузеру, чтобы он установил для него значение по умолчанию и скрол вернулся на страницу.

Вариант с .toggle() :

Т.к. для модального окна <div class="modal"> в css изначально установлен display: none, то заменив строчки 104-105 и 11-112 на переключение (добавление если нет/удаление если есть) класса .show можно получить тот же функционал:

 modal.classList.toggle('show');

Реализация функционала, чтобы модальное окно закрывалась по клику на подложку(тёмно-прозрачная) или по нажатию ESC.

Закрытие по клику на подложку:  
Т.к. функционал по закрытию модального она уже прописывался для клика по «крестику» в строках 110-113, выносим его в отдельную функцию, чтобы его было проще использовать повторно:

function closeModal() {

        modal.classList.add('hide');

        modal.classList.remove('show');

        document.body.style.overflow = '';

    }

Обработчику события клика на «крестик» передаём эту функцию:

modalCloseBtn.addEventListener('click', closeModal);

Логика реализации закрытия при клике по подложке следующая: если клик происходит по «белой области», то он попадает в .modal\_\_dialog – это обёртка для контента модального окна, окно закрываться не должно. Но вокруг него находится элемент-родитель с классом .modal, это и есть подложка, при клике по которой окно должно закрыться. Следовательно нужно отследить куда кликнул пользователь – event.target, если по .modal, то закрыть окно.

modal.addEventListener('click', (e) => {

        if (e.target === modal) {

            closeModal();

        }

    });

На элемент с классом .modal вещается обработчик события клика. Если целью клика был элемент с классом .modal(подложка), вызывается функция closeModal() закрывающая модальное окно.

Закрытие по нажатию ESC:

Существует событие keydown срабатывающее тогда, когда нажимается кнопка. Так же необходим объект события event, у которого есть свойство code, отслеживающее код нажатой клавиши, если это ESC то нужно закрыть окно.

document.addEventListener('keydown', (e) => {

        if (e.code === 'Escape' && modal.classList.contains('show')) {

            closeModal();

        }

    });

Глобальному объекту document назначается обработчик события по нажатию клавиши. Проверяется условие, если в объекте события код нажатой клавиши соответствует 'Escape' И список классов модального окна содержит класс .show, т.е. окно отображается то вызывается функция closeModal() закрывающая модальное окно. Догадка: вторая проверка необходима т.к. обработчик навешивается прямо на document и чтобы closeModal() не запускалась каждый раз при нажатии ESC на странице, лучше проверить открыто ли модальное окно вообще.

044 МОДИФИКАЦИИ МОДАЛЬНОГО ОКНА (Практика 10)

Задача: модальное окно должно появляться, когда пользователь долистал станицу до конца или через определённый промежуток времени. Очень частая задача!

1. Решаем вопрос появления модального окна через 15 сек.

Для того, чтобы вызвать модальное окно после определённого промежутка времени, необходимо передать в setTimeout либо функционал по вызову модального окно, либо готовую функцию.

Функционал вызова модального окна передан обработчику событий и срабатывает по клику на каждый элемент псевдомассива в переменной modalTrigger. Выносим этот функционал за рамки обработчика в отдельную функцию openModal:

function openModal() {

        modal.classList.add('show');

        modal.classList.remove('hide');

        document.body.style.overflow = 'hidden';

    }

    modalTrigger.forEach(item => {

        item.addEventListener('click', openModal);

    });

const modalTimerId = setTimeout(openModal, 15000);

Создаётся переменная modalTimerId в которую помещается вызов функции openModal через 15 секунд.

Для того, чтобы модальное окно не появлялось через 15, в случае если пользователь ранее уже вызывал его кликами по соответствующим кнопкам, дополняем функцию openModal строкой clearInterval(modalTimerId):

function openModal() {

        modal.classList.add('show');

        modal.classList.remove('hide');

        document.body.style.overflow = 'hidden';

clearInterval(modalTimerId);

    }

Смысл: функция openModal передаётся обработчику событий кликов на кнопках. Если пользователь кликнул по кнопке до того как сработал setTimeout(openModal, 15000), повторный вызов функции через 15 сек отменяется.

1. Вызов модального окна по достижению конца страницы.

Будем использовать событие scroll навешиваемое на глобальный объект window.

Чаще всего скрипты пишутся по принципу сначала рабочий вариант, затем оптимизированный.

   function showModalByScroll() {

        if (window.pageYOffset + document.documentElement.clientHeight >=

document.documentElement.scrollHeight) {

openModal();

            window.removeEventListener('scroll', showModalByScroll);

        }

    }

    window.addEventListener('scroll', showModalByScroll);

Объявляется функция showModalByScroll, которая будет производить сравнение - если window.pageYOffset (свойство объекта window содержащее количество пикселей на которое страница прокручена по вертикали/невидимая часть) + document.documentElement.clientHeight (высота видимой области окна/видимая часть) >= document.documentElement.scrollHeight (полный размер страницы с учётом прокрутки). Как только это условие срабатывает, это означает, что пользователь долистал страницу до конца, вызывается функция openModal открывающая модальное окно.

Глобальному объекту window навешивается обработчик события 'scroll', а в качестве callback функции передаётся вышеописанная showModalByScroll.

Для того, чтобы модальное окно не открывалось постоянно при каждом достижении конца страницы, внутри функции showModalByScroll после вызова функции открывающей окно openModal, прописывается удаление обработчика событий removeEventListener которому нужно передать именно ту же функцию/не анонимную!, иначе удаление не сработает!

ОТКЛЮЧЕНИЕ КНОПКИ

В html кнопку можно отключить установив атрибут disabled в положение disabled disabled = “disabled”.

login.setAttribute('disabled', 'disabled');

Кнопке(ам) с помощью метода .setAttribute атрибуту disabled устанавливается значение disabled.

let form = document.querySelector('#loginForm'),

    userName = document.querySelector('#username'),

    password = document.querySelector('#password'),

    login = document.querySelector('#loginBtn');

form.addEventListener('input', () => {

    if (userName.value.length > 0 &&

        password.value.length > 0) {

        login.removeAttribute('disabled');

    } else {

        login.setAttribute('disabled', 'disabled');

    }

});

Кнопка Login по умолчанию выключена. Со страницы получаем элементы: форму, инпуты (юзернейм и пароль) и кнопку. Форме назначается обработчик событий, в качестве события указывается ‘input’ задаётся условие если длина введённых пользователем данных больше 0 (или другого заданного значения) метод .removeAttribute удаляет у кнопки атрибут disabled. Иначе .setAttribute устанавливает disabled в значение disabled.

045 ФУНКЦИИ-КОНСТРУКТОРЫ

КОНСТРУКТОРЫ И КЛАССЫ

При создании функции в JS, создаётся новый объект, свойства которого уже прописаны в этой функции.

Функции-конструкторы используются для конструирования объектов и создания множества подобных копий. Примеры: создание новых пользователей сайта, товаров в магазине, роликов на Youtube, даже компонентов сайтов таких как слайдеры т.е. всего где можно использовать один шаблон.

Конструкторы использовались в стандарте ES5, в ES6 используются классы (классы считаются «синтаксическим сахаром»).

Синтаксис функции-конструктора/конструктора:

function User(name, id) {

*this*.name = name;

*this*.id = id;

*this*.human = true;

*this*.hello = function() {

        console.log("Hello " + *this*.name);

    };

}

User.prototype.exit = function(name) {

    console.log('Пользователь ' + *this*.name + ' вышел');

};

Все методы и свойства прописанные в конструкторе будут присутствовать в каждом созданном с помощью него объекте.

С помощью свойства .prototype можно добавлять новые методы и свойства в конструктор и они будут прототипно наследоваться у потомков. Этот приём используется когда нет доступа к прототипу или его нельзя менять по каким-то причинам, но его нужно немного модифицировать (скорее всего речь идёт о задании прототипа с помощью Object.setPrototypeOf() или об устаревшем .\_\_proto\_\_).

Синтаксис класса:

class User {

   constructor(name, id) {

*this*.name = name;

*this*.id = id;

*this*.human = true;

    }

    hello () {

        console.log(`Hello! ${*this*.name}`);

    }

    exit () {

        console.log(`Пользователь ${*this*.name} вышел`);

    }

}

КОНТЕКСТ ВЫЗОВА (this)

this – это то, что окружает функцию и в каких условиях она вызывается.

Функция может вызываться 4мя способами и в каждом случае контекст вызова будет отличаться.

1. Простой вызов функции

function showThis() {

    console.log(*this*);

}

showThis();

Контекстом вызова для этой функции является(ссылается на) объект Window (глобальный объект в вэб браузере). Верно для работы без строгого режима.

'use strict';

function showThis(a, b) {

    console.log(*this*);

    function sum() {

        console.log(*this*);

        return a + b;

    }

    console.log(sum());

}

showThis(4, 5);

В консоли: script.js:4 undefined

script.js:7 undefined

script.js:11 9

Если объявлен use strict(появился в ES6), функции не могут себя выполнить(у них нет своего this) вместо Window выдают undefined – не могут найти свой контекст вызова. Функция sum вернёт сумму из замыкания.

1. Использование методов объектов(они так же являются функциями)

let obj = {

    a: 20,

    b: 15,

    sum: function() {

        console.log(*this*);

    }

};

obj.sum();

В консоли: {a: 20, b: 15, sum: ƒ}

a: 20

b: 15

sum: ƒ ()

\_\_proto\_\_: Object

Вызов метода sum объекта obj даёт(ссылается на) сам объект. Контекст выполнения методов объекта – сам объект.

let obj = {

    a: 20,

    b: 15,

    sum: function() {

        console.log(*this*);

        function onceAgain() {

            console.log(*this*);

        }

        onceAgain();

    }

};

obj.sum();

Для того чтобы функция onceAgain выполнилась, её необходимо предварительно вызвать(строка 9). Контекстом выполнения функции onceAgain будет глобальный объект Window или undefined если ‘use strict’, т.к. она НЕ является методом объекта obj, а является функцией внутри функции.

1. Использование функции через new

Когда создаётся новый объект(с помощью конструктора), контекст вызова для всех методов и свойств будет только что созданный объект, т.е. this ссылается на новосозданный объект.

function User(name, id) {

*this*.name = name;

*this*.id = id;

*this*.human = true;

*this*.hello = function() {

        console.log("Hello " + *this*.name);

    };

}

const ivan = new User('Ivan', 20);

В данном случае контекстом вызова будет только что созданный объект ivan. this в конструкторах и классах – это новый экземпляр объекта.

1. Указание конкретного контекста (ручное присваивание this любой функции)

.call, .apply, .bind.

let user = {

    name: 'John'

};

function sayName(surname) {

    console.log(*this*);

    console.log(*this*.name + surname);

}

console.log(sayName.call(user, 'Smith'));

console.log(sayName.apply(user, ['Snow']));

Изначально функция sayName никак не связана с объектом user, т.е. this ссылается на глобальный объект Window или undefined. Но использование метода .call(если в качестве дополнительного(surname) параметра передаётся одна строка) и метода .apply(если в качестве дополнительного(surname) параметра нужно передать массив из нескольких значений) присваивают this значение объекта, который был передан им в качестве аргумента. Т.е. благодаря этим методам функция sayName приобрела свой контекст вызова в виде объекта user.

function count(number) {

    return *this* \* number;

}

let double = count.bind(2);

console.log(double(3));

.bind() создаёт новую! функцию связанную с определённым контекстом. В переменную double помещается новая функция у которой есть жестко привязанный контекст! Метод .bind() передаёт функции count своё содержимое в качества контекста вызова (в данном случае this = 2). Считается более «жестким» методом привязки контекста, используется не часто на нативном JS. Зато часто используется в React (с помощью этого метода другие методы жестко связываются с определёнными классами).

Работа контекста вызова в DOM:

const btn = document.querySelector('button');

btn.addEventListener('click', function() {

    console.log(*this*);

*this*.style.width = 200 + 'px';

    function showThis() {

        console.log(*this*);

    }

    showThis();

});

Если в обработчиках событий используется контекст вызова this и используется Обычное объявление функции(не стрелочное!) обработчика, то контекстом вызова будет тот элемент на котором происходит событие (тут this = event.target, но чаще используется event.target). В примере контекст вызова это <button>, при клике её ширина станет 200px. НО если объявить функцию внутри функции-обработчика (showThis) и вызвать, то её контекстом вызова останется глобальный объект Window или undefined.

btn.addEventListener('click', (e) => {

    e.target.style.width = 200 + 'px';

});

Если callback функция обработчика событий задана как стрелочная, то для избегания ошибок лучше работать с event.target чем с this.

Пример со стрелочной функцией №1:

const obj = {

    num: 5,

sayNumber: function() {

        const say = () => {

            console.log(*this*);

        };

        say();

    }

};

obj.sayNumber();

Особенность стрелочной функции в том, что у неё нет своего контекста вызова, она возвращает контекст вызова родителя, родителем является метод sayNumber, а у метода контекст ссылается на объект в котором он существует - объект obj. Следовательно при вызове метода .sayNumber() в консоль будет выведен сам объект { num: 5, sayNumber: [Function: sayNumber] }. Если вместо стрелочной была бы обычная функция, то контекстом вызова был бы глобальный объект Window или undefined.

Пример со стрелочной функцией №1:

Стрелочные функции часто используются для модификации элементов прямо «наместе».

const double = (a) => {

    return a \* 2;

};

Если выполняется действие в одну строку и результат возвращается, то можно переписать как :

const double = (a) => a \* 2;

Если аргумент только один, то можно убрать круглые скобки:

const double = a => a \* 2;

047 КЛАССЫ (ES6)

Классы – это красивая обёртка функций-конструкторов. Название классов как и конструкторов всегда начинается с большой буквы.

class Rectangle {

    constructor(height, width) {

*this*.height = height;

*this*.width = width;

    }

    calcArea() {

        return *this*.height \* *this*.width;

    }

}

const square = new Rectangle(10, 10);

Для «конструирования» класса используется свойство constructor. Аргументами для constructor будут те значения, которые будут приходить извне, в примере height, width. Затем прописываются свойства будущих объектов/экземпляров. Следом прописываются методы. После фигурных скобок {} конструктора и между методами ; не ставится! Метод начинается с имени, без :, без слова function. В переменную square помещается объект являющийся экземпляром класса Rectangle.

Принципы ООП:

1. Абстракция – отделяем концепцию от её экземпляра. Концепция – класс, экземпляр – экземпляр класса.
2. Наследование – способность объекта или класса базироваться на другом объекте или классе. Главный механизм для повторного использования кода. Наследственное отношение классов чётко определяет их иерархию.

class ColoredRectangleWithText extends Rectangle {

    constructor(height, width, text, bgColor) {

*super*(height, width);

*this*.text = text;

*this*.bgColor = bgColor;

    }

    showMyProps() {

        console.log(`Текст: ${*this*.text}, цвет: ${*this*.bgColor}`);

    }

}

const div = new ColoredRectangleWithText(25, 10, 'Hello!', 'red');

Ключевое слово extends говорит о том, что класс ColoredRectangleWithText будет наследоваться от класса Rectangle. В constructor должны передаваться те же аргументы, что и в классе Rectangle + новые аргументы, которые присутствуют только в классе ColoredRectangleWithText. Чтобы не переписывать те же свойства в «новый» конструктор, используется метод super(), который вызывает суперконструктор родителя. super() всегда! должен быть на первом месте в конструкторе. Аргументами передаваемыми в super() будут те свойства, которые будут необходимы в «новом» конструкторе, т.к. не все свойства родителя всегда необходимы. Затем прописываются свойства и методы наследующего/«нового» класса. В переменную square помещается объект являющийся экземпляром класса ColoredRectangleWithText обладающий в том числе свойствами и методами класса Rectangle.

048 ИСПОЛЬЗУЕМ КЛАССЫ В РЕАЛЬНОЙ РАБОТЕ (Практика 11)

Задача: создать карточки с вариантами продуктовых меню не с помощью вёрстки, а с помощью класса.

Создаётся класс -шаблон, экземплярами которого будут карточки меню. Конструктору передаются аргументы, которые необходимы для создания карточки: src – путь к картинке, alt – альтернативное описание картинки в тэге <img>, title – заголовок карточки, descry – описание, price – цена. Смысл создания карточек через класс в том, что в дальнейшем аргументы могут приходить с сервера и карточки будут формироваться динамически. Т.к. цена будет приходить в $, а цена на сайте в грн перед помещением на страницу её необходимо конвертировать. В классе прописывается свойство this.transfer хранящее текущий курс(который в дальнейшем так же может откуда-то приходить как один из агрументов) и метод для конвертации валют changeToUAH(). Для того, чтобы в карточку на страницу передавалось конвертированное значение, метод changeToUAH() нужно вызывать либо внутри метода, который будет что-то помещать на страницу ( render() ) либо прямо в конструкторе, это тоже допустимо.

Метод render() должен создавать элемент, помещать в него определённую вёрстку, вёрстку дополнить данными, которые приходят в качестве аргументов в конструктор и поместить этот элемент на страницу. Вёрстку можно заранее прописать в index.html, перенести в script.js и оставить только «каркас» очистив от наполнения, которое будет приходить из аргументов класса. Затем используя свойство .innerHTML вёрстка подставляется её в элемент/карточку на странице.

Чтобы поместить элемент на страницу, необходимо получить его родителя – куда будет вставлен <div> с карточками. Для этого в конструктор добавляется новый аргумент parentSelector, затем прямо в конструкторе прописывается получение родительского элемента (свойство так же можно вычислять на этапе метода render() ). После в методе render() в родительский элемент this.parent с помощью метода .append() помещается новосозданный.

class MenuCard {

        constructor(src, alt, title, descr, price, parentSelector) {

*this*.scr = src;

*this*.alt = alt;

*this*.title = title;

*this*.descr = descr;

*this*.price = price;

*this*.parent = document.querySelector(parentSelector);

*this*.transfer = 27;

*this*.changeToUAH();

        }

        changeToUAH() {

*this*.price = *this*.price \* *this*.transfer;

        }

        render() {

            const element = document.createElement('div');

            element.innerHTML = `

                <div class="menu\_\_item">

                        <img src=${*this*.scr} alt=${*this*.alt}>

                        <h3 class="menu\_\_item-subtitle">${*this*.title}</h3>

                        <div class="menu\_\_item-descr">${*this*.descr}</div>

                        <div class="menu\_\_item-divider"></div>

                        <div class="menu\_\_item-price">

                            <div class="menu\_\_item-cost">Цена:</div>

                            <div class="menu\_\_item-total"><span>${*this*.price}</span> грн/день</div>

                        </div>

                </div>

            `;

*this*.parent.append(element);

        }

    }

Далее необходимо создать новый объект/экземпляр класса MenuCard и вызвать метод .render() который отвечает за формирование/рендеринг карточки на странице:

new MenuCard(

        "img/tabs/vegy.jpg",

        "vegy",

        'Меню "Фитнес"',

        'Меню "Фитнес" - это новый подход к приготовлению блюд: больше свежих овощей и фруктов. Продукт активных и здоровых людей. Это абсолютно новый продукт с оптимальной ценой и высоким качеством!',

        9,

        '.menu .container'

    ).render();

new MenuCard(

        "img/tabs/elite.jpg",

        "elite",

        'Меню “Премиум”',

        'В меню “Премиум” мы используем не только красивый дизайн упаковки, но и качественное исполнение блюд. Красная рыба, морепродукты, фрукты - ресторанное меню без похода в ресторан!',

        14,

        '.menu .container'

    ).render();

new MenuCard(

        "img/tabs/post.jpg",

        "post",

        'Меню "Постное"',

        'Меню “Постное” - это тщательный подбор ингредиентов: полное отсутствие продуктов животного происхождения, молоко из миндаля, овса, кокоса или гречки, правильное количество белков за счет тофу и импортных вегетарианских стейков.',

        21,

        '.menu .container'

    ).render();

Объект не обязательно помещать в переменную, это допустимо тогда, когда этот объект используется на месте. Если объект не сохранён в переменную, он создастся и удалится, сослаться на него будет невозможно.

Для того, чтобы использовать объект и метод на месте, прописывается new MenuCard().render(), а не const card = new Menu Card(); card.render().

Далее внутрь класса передаются необходимые аргументы, благодаря которым карточки будут отличаться. В данном случае в качестве аргументов копируются те значения, которые были в вёрстке. Аргументы передаются в том виде, в котором они будут вставлены в вёрстку, т.е. с кавычками, для строк с кавычками внутри, использовать комбинацию кавычек ('Меню "Постное"'). Для того, чтобы вёрстка не сломалась, необходимо удалить/закомментировать карточки, которые были в вёрстке, чтобы остались только сгенерированные с помощью JS.

Стандарт ES6 (наиболее часто используемые)

049 Rest оператор и параметры по умолчанию (ES6) (Практика 12)

Интерполяция:

ES6 был выпущен в 2015 году.

Интерполяция – более простой способ вставить переменную или выражение. Для её применения используются кавычки находящиеся на клавише ~ Ё, а перед переменными ставится знак $, сама переменная помещается в {}. Приём интерполяции постоянно используется в современном коде.

let name = 'Ivan',

    age = '30',

    mail = 'ex@mail.ru';

document.write(`Пользователю ${name} ${age} лет. Его почтовый адрес ${mail}`);

Уточнение по созданию переменных(var, let, const) – При использовании let или const в цикле для каждой итерации создаётся своя переменная! Если в цикле используется объявление i(итерации) через var, то i остаётся одинаковой во всём цикле, в каждой итерации она не создаётся заново. Если нужно, чтобы i сбрасывалась, она объявляется через let.

Стрелочные функции, особенности:

- Более лаконична;

- Анонимна, её можно только заранее поместить в какую-то переменную и потом вызвать(т.е. функциональное выражение);

let fun = () => {

    console.log(*this*);

};

fun();

- Не сможем управлять обработчиками событий, если это необходимо (хз что значит);

- Её нельзя запустить внутри себя (сделать рекурсию);

- НЕ ИМЕЕТ своего контекста вызова this, НО получает его от своего родителя:

let obj = {

    number: 5,

    sayName: function() {

        let say = () => {

            console.log(*this*);

        };

        say();

    }

};

obj.sayName();

Возвращает объект obj, если была бы обычная функция, не стрелочная, возвращала бы глобальный объект Window или undefined.

Чаще всего стрелочные функции используются в обработчиках событий, setInterval, setTimeout и в технологии Ajax.

let btn = document.querySelector('button');

btn.addEventListener('click', function() {

    let show = () => {

        console.log(*this*);

    };

    show();

});

Вернёт элемент <button>, т.к. у стрелочной функции нет своего контекста вызова, а контекст вызова функции-родителя – это элемент <button>.

Параметры по умолчанию:

Актуально в ES5:

function calcOrDouble(number, basis) {

    basis = basis || 2;

    console.log(number \* basis);

}

calcOrDouble(3, 5);

calcOrDouble(6);

Для basis параметром по умолчанию будет 2 (строка 2), если не задано другое значение. Выполнение строки 5 даст 15 (number=3, basis=2), а строки 6 даст 12 (number=6, basis=2(default).

Для ES6:

function calcOrDouble(number, basis = 2) {

    console.log(number \* basis);

}

calcOrDouble(3, 5);

calcOrDouble(6);

Значение по умолчанию задаётся прямо в параметрах функции. Результат работы тот же.

Параметры по умолчанию используются, к примеру, для задания первоначальных высоты и ширины модального окна, цвета текста, цвета заднего фона.

Классы:

class Rectangle {

    constructor(height, width) {

*this*.height = height;

*this*.width = width;

    }

    calcSquare() {

        return *this*.height \* *this*.width;

    }

}

const square = new Rectangle(10, 10);

console.log(square.calcSquare());

Классы используются вместо конструкторов, в данном случае был создан класс с методом считающим площадь, его потомок наследует этот метод от класса, в консоль выводится 100. Классы используются для создания новых пользователей, элементов интерфейса, модальных окон и т.д.

Spread оператор(оператор разворота):

Он разворачивает(раскрывает) массив и превращает его просто в набор данных, выглядит как три точки …

let video = ['youtube', 'vimeo', 'rutube'],

    blogs = ['wordpress', 'livejournal', 'blogger'],

    internet = [...video, ...blogs, 'vk', 'facebook'];

console.log(internet);

(8) ["youtube", "vimeo", "rutube", "wordpress", "livejournal", "blogger", "vk", "facebook"]

Выдаст объединённый массив из 8ми элементов, а не массив из 4х элементов(2х массивов и 2х строк).

function log(a, b, c) {

    console.log(a);

    console.log(b);

    console.log(c);

}

let numbers = [2, 5, 7];

log(...numbers);

Благодаря spread оператору(…) функция log получает в качестве аргументов не массив а 3 отдельных значения, с которыми может дальше работать.

Rest оператор:

Оператор rest объединяет разрозненные значения/элементы в один массив( те же три точки …):

const log = function(a, b, ...rest) {

    console.log(a, b, rest);

};

log('first', 'second', 'third', 'fourth');

В консоли: first second [ 'third', 'fourth' ].

Оператор rest всегда передаётся функции последним, но может быть назван как угодно, главное чтобы перед названием стояли …

Может быть использован в функциях, когда, к примеру, неизвестно сколько ещё будет аргументов.

Функция log в примере вызывается с 4мя переданными аргументами, первые два, соответствующие a и b выводятся в консоль по отдельности, остальные (rest) не зависимо от количества объединяются в массив.

Практика 12

Задача: в методе .render() класса MenuCard карточки помещаются в лишний <div>, а должны помещаться в <div class="menu\_\_item">, необходимо это исправить.

Для выполнения такой задачи нужно новосозданному <div> присвоить класс class="menu\_\_item".

Дополнительные задачи: предусмотреть механизм дополнительной кастомизации карточки, предусмотреть css класс по умолчанию.

Для реализации механизма используем rest оператор и параметры по умолчанию.

class MenuCard {

        constructor(src, alt, title, descr, price, parentSelector, ...classes) {

*this*.scr = src;

*this*.alt = alt;

*this*.title = title;

*this*.descr = descr;

*this*.price = price;

*this*.parent = document.querySelector(parentSelector);

*this*.classes = classes;

*this*.transfer = 27;

*this*.changeToUAH();

        }

        changeToUAH() {

*this*.price = *this*.price \* *this*.transfer;

        }

        render() {

            const element = document.createElement('div');

            if (*this*.classes.length === 0) {

*this*.element = 'menu\_\_item';

                element.classList.add(*this*.element);

            } else {

*this*.classes.forEach(className => element.classList.add(className));

            }

            element.innerHTML = `

                <img src=${*this*.scr} alt=${*this*.alt}>

                <h3 class="menu\_\_item-subtitle">${*this*.title}</h3>

                <div class="menu\_\_item-descr">${*this*.descr}</div>

                <div class="menu\_\_item-divider"></div>

                <div class="menu\_\_item-price">

                    <div class="menu\_\_item-cost">Цена:</div>

                    <div class="menu\_\_item-total"><span>${*this*.price}</span> грн/день</div>

                </div>

            `;

*this*.parent.append(element);

        }

    }

    new MenuCard(

        "img/tabs/vegy.jpg",

        "vegy",

        'Меню "Фитнес"',

        'Меню "Фитнес" - это новый подход к приготовлению блюд: больше свежих овощей и фруктов. Продукт активных и здоровых людей. Это абсолютно новый продукт с оптимальной ценой и высоким качеством!',

        9,

        '.menu .container',

        'menu\_\_item'

    ).render();

    new MenuCard(

        "img/tabs/elite.jpg",

        "elite",

        'Меню “Премиум”',

        'В меню “Премиум” мы используем не только красивый дизайн упаковки, но и качественное исполнение блюд. Красная рыба, морепродукты, фрукты - ресторанное меню без похода в ресторан!',

        14,

        '.menu .container',

        'menu\_\_item'

    ).render();

    new MenuCard(

        "img/tabs/post.jpg",

        "post",

        'Меню "Постное"',

        'Меню “Постное” - это тщательный подбор ингредиентов: полное отсутствие продуктов животного происхождения, молоко из миндаля, овса, кокоса или гречки, правильное количество белков за счет тофу и импортных вегетарианских стейков.',

        21,

        '.menu .container',

        'menu\_\_item'

    ).render();

К аргументам конструктора добавляется аргумент …classes (с оператором rest). Затем аргумент classes прописывается в свойства, чтобы его можно было в дальнейшем использовать this.classes = classes, это массив и работать с ним нужно как с массивом. Функционал метода .render() расширяется для работы с новым массивом. Нужно перебрать все элементы внутри массива classes, «вытащить» названия css классов и добавить их к <div> в котором находятся карточки: this.classes.forEach(className => element.classList.add(className));. Затем необходимо убрать обёртку из <div class="menu\_\_item">, т.к. этот класс будет присваиваться новосозданному <div>.

После при создании карточек/экземпляров класса MenuCard последним аргументом передаётся класс menu\_\_item, который необходимо добавить <div>.

Задание css класса по умолчанию способом ES6 (...classes = ’menu\_\_item’) выдаёт ошибку, т.к. rest параметр не поддерживает дефолтные значения. Так же не сработает способ ES5 this.classes = classes || ’menu\_\_item’, т.к. далее идёт перебор массива методом .forEach() и даже если предыдущий код сработает, у строки нет этого метода и дальше код не выполнится. Поэтому будут использованы обычные условия, прописанные внутри метода .render(). Если длинна массива с css классами, находящегося внутри classes = 0, т.е. внутри него нет ни одного класса. То будет создано новое свойство this.element класса MenuCard (element – условное имя, может называться как угодно, это не созданный выше <div>), в которое будет помещена строка с именем css класса ’menu\_\_item’. (эта строка написана на случай, если вдруг этот класс ещё где-то понадобится). Затем <div> у в переменной element добавляется класс ’menu\_\_item’. Иначе, т.е. если массив внутри classes содержит какие-то классы, они будут добавлены element описанным выше методом с .forEach().

ЛОКАЛЬНЫЙ СЕРВЕР

http запросы: get – получить информацию с сервера(поиск через гугл, кнопка «загрузить ещё» на сайте).post – отправляет информацию на сервер.

Простые сервера(типа liveserver, browsersync) поддерживают только get запросы. Для полноценной работы нужны локальные сервера типа MAMP или OpenServer.

Технология Ajax – серверная технология, позволяет отправлять get и post запросы без перезагрузки страницы.

Сервер принимает данные в JSON формате.

JSON ФОРМАТ ПЕРЕДАЧИ ДАННЫХ, ГЛУБОКОЕ КЛОНИРОВАНИЕ ОБЪЕКТОВ

Расшифровывается как JavaScript object notation, является текстовым форматом обмена данными.

Используется не только для передачи но и для хранения данных.

Изначально формат появился в JS, но сейчас может использоваться практически любым яп. Представляет из себя набор пар ключ: значение. ГЛАВНОЕ ПРАВИЛО – все строки в двойных кавычках “”. Значениями могут быть объекты, массивы, числа, строки, логические значения или null.

Для работы с данными на сервере у JSON есть 2 метода их преобразования:

let options = {

    width: 1366,

    height: 768,

    background: 'red',

    font: {

        size: '16px',

        color: '#fff'

    }

};

console.log(JSON.stringify(options));

// Результат:

{"width":1366,"height":768,"background":"red","font":{"size":"16px","color":"#fff"}}

Метод stringify преобразовывает содержимое объекта в одно строку(формата JSON) , всё свойства записаны в двойных кавычках, даже там где были использованы одинарные.

let options = {

    width: 1366,

    height: 768,

    background: 'red',

    font: {

        size: '16px',

        color: '#fff'

    }

};

console.log(JSON.parse(JSON.stringify(options)));

Метод parse выполняет обратное преобразование из формата JSON в формат JS. В данном случае в консоли будет вышеуказанный объект options.

Напрямую передать объект на сервер нельзя, т.к. сервер и протоколы передачи данных этого не поймут, поэтому его необходимо преобразовать в один из вариантов, которые можно передавать и уже после этого передать данные с фронтенда на бэкенд.

Именно при помощи JSON формата часто клиентская часть общается с сервером.

Основные преимущества JSON формата – маленький вес файлов и простота чтения. До его появления использовался язык XML.

Глубокие копии объектов:

Создаются при помощи JSON, синтаксис:

const options = {

    width: 1366,

    height: 768,

    background: 'red',

    font: {

        size: '16px',

        color: '#fff'

    }

};

const clone = JSON.parse(JSON.stringify(options));

Объяснение: JSON.stringify(options) превратит объект options в формат JSON, JSON.parse(… возьмёт

формат JSON и «распарсит» его назад в объект формата JS. Созданный объект будет помещён в переменную clone, это будет глубокая копия, которая не зависит от первоначального объекта.

Доказательство:

clone.font.size = '25px';

console.log(options);

console.log(clone); //В консоли:

{

  width: 1366,

  height: 768,

  background: 'red',

  font: { size: '16px', color: '#fff' }

}

{

  width: 1366,

  height: 768,

  background: 'red',

  font: { size: '25px', color: '#fff' }

}

AJAX И ОБЩЕНИЕ С СЕРВЕРОМ

Asynchronous JavaScript and XML Данная технология не перезагружает страницу каждый раз, а перезагружает только выбранную часть(фильтры на сайте), так же происходит ускорение реакций интерфейса(в ходе набора, предлагаются поисковые запросы). Уменьшается нагрузка на сервер и экономится трафик пользователя.

Недостатки: у пользователя должен быть включен JS, при плохом интернете может быть некорректное поведение, до 2017 года была проблема с СЕО оптимизацией(поисковые системы не находили такой контент).

Для того чтобы страница могла общаться с сервером нужны http запросы, которые будут отправляться. Можно отправлять данные, постить данные и выполнять другие операции. Для того, чтобы делать это асинхронно, без перезагрузки страницы нужен AJAX.

Первый вариант AJAX’a(не совсем актуален):

Для того, чтобы из JS делать http запросы к серверу без перезагрузки страницы, нужен встроенный объект XMLHttpRequest. У него есть свои методы, свойства и события.

let request = new XMLHttpRequest();

сохранение в переменную request результата создания конструктором и последующего вызова объекта. Т.к. это конструктор, можно создать сколько угодно таких запросов, при этом каждый раз он может вести себя по разному и реализовываться в разных условиях.

Методы XMLHttpRequest:

request.open();

1)Метод open чаще всего идёт сразу за созданием объекта XMLHttpRequest, производит/собирает настройку AJAX запроса. Может принимать 5 различных аргументов:

method – метод по которому клиент общается с сервером(чаще всего get или post);

url – путь к серверу(локальный файл или облачный/из сети). Запросы посылаются исходя из расположения html файла, поэтому путь формируется относительно index.html;

async – асинхронность запроса, по умолчанию true, если указать false то пока сервер не ответит со страницей нельзя будет взаимодействовать(зависнет). Синхронность – выполнение по порядку, если операция выполняется долго, остальной код ждёт, асинхронность – выполнение не зависимо от выполнения остального кода, асинхронный код не блокирует остальной! AJAX запросы по умолчанию асинхронный код – запрос посылается на сервер, остальной код продолжает выполняться не ожидая ответа сервера;

login – логин;

pass – пароль.

request.open('GET', 'js/current.json');

настроенный запрос к серверу – указаны метод и url адрес файла .json(путь относительно index.html).

HTTP запросы:

Обычно состоят из двух частей – head и body.

Когда AJAX запрос настроен, можно его до настроить – нужно указать что именно отправляется, что это за информация, в чём закодирована и т.д. Это делается для того, чтобы трансферные протоколы чётко понимали что они передают, и когда запрос доходит до сервера, он тоже понимал что он принимает в себя(JSON файл, картинки или что-то ещё). Для таких целей существуют http заголовки, ниже будет использован заголовок для передачи JSON файлов:

request.setRequestHeader('Content-type', 'application/json; charset=utf-8');

2)Метод setRequestHeader используя объект XMLHttpRequest, находящийся в переменной request, применяется для настройки http запросов. В частности, указывается какой контент будет(Content-type), а это будет json в кодировке utf-8.

После таких настроек уже можно отправить запрос:

request.send(body);

3)Метод send открывает соединение и отправляет запрос на сервер. В качестве аргумента, он принимает body – тело HTTP запроса/данные которые уходят на сервер, НО тело у запросов бывает только тогда, когда информация берется из клиентской части и отправляется на серверную(пример – форма обратной связи на сайте). body существует только в методах POST и некоторых других, в GET body не будет.

request.send();

без аргумента send просто запускает запрос и запрос идёт за ответом к серверу.

Свойства XMLHttpRequest:

Суть свойств – получаем ответ от сервера и что-то с ним(ответом) делаем.

1)status – содержит http код ответа сервера (404, 0, 203, 200 – всё ок и т.д.)

2)statusText – содержит текстовое описание ответа от сервера(ok, not found)

3)responseText / response – содержит текст ответа сервера/ответ от сервера который задал backend (тот, что нужно использовать на клиенте).

4)readyState – возвращает текущее состояние запроса:

- 0 UNSENT – объект XMLHttpRequest создан, метод open() ещё не вызывался.

- 1 OPENED – метод open() вызван

- 2 HEADERS\_RECEIVED – метод send() был вызван, доступны заголовки и статус

- 3 LOADING – загрузка; responseText содержит частичные данные

- 4 DONE – Операция полностью завершена

Все состояния можно отслеживать, но чаще всего отслеживается 4.

События XMLHttpRequest:

loadStart – начало загрузки запроса;

progress – выполнение запроса;

abort – отмена выполнения;

timeOut – время выполнения вышло(обычно > 30 сек);

loadEnd – завершение загрузки запроса.

Используются часто:

onreadystatechange – отслеживает статут готовности запроса в данный момент(следит за изменением свойства readyState и срабатывает при каждом изменении его состояния).

load – срабатывает когда запрос полностью загрузился без ошибок и получен какой-то результат.

События используются для навешивания обработчиков событий.

const inputRub = document.querySelector('#rub'),

    inputUsd = document.querySelector('#usd');

inputRub.addEventListener('input', () => {

    const request = new XMLHttpRequest();

    request.open('GET', 'js/current.json');

    request.setRequestHeader('Content-type', 'application/json; charset=utf-8');

    request.send();

    request.addEventListener('readystatechange', () => {

        if (request.readyState === 4 && request.status === 200) {

            const data = JSON.parse(request.response);

            inputUsd.value = (+inputRub.value / data.current.usd).toFixed(2);

        } else {

            inputUsd.value = 'Что-то пошло не так!';

        }

    });

});

По id со страницы получаются инпуты и помещаются в переменные inputRub и inputUsd. Переменной inputRub назначается обработчик событий по событию input(ввод данных в форму). Объявляется переменная request в которую помещается результат создания конструктором и вызова встроенного в браузер объекта XMLHttpRequest. Объект инициализируется методом open(),в него передаются параметры: метод GET и url адрес куда отправлять запрос. Строка 6, методом setRequestHeader() устанавливается значение заголовка (header) http запроса, имя (name) Content-Type, значение (value) application/json... Строка 7, метод send() устанавливает соединение и отправляет запрос серверу. Строка 8, после запроса экземляру объекта XMLHttpRequest в переменной request назначается обработчик событий по событию readystatechange(изменения в состоянии объекта). Задаётся условие, что при соответствии свойства readyState состоянию 4 И свойства status (код состояния) значению 200, т.е. всё ОК и можно использовать response – ответ от сервера(ожидаем получить объект JSON

{

"current": {

"usd": 74

}

}), в переменную data будет помещён объект полученный в результате преобразования ответа сервера из формата JSON в формат JS. Строка 11 вычисляется результат деления значения (value) инпута inputRub на значение свойства usd объекта в переменной дата, результат вычисления будет помещён в значение инпута inputUsd.value. Метод toFixed(2) указывает на количество знаков после запятой. Если условие не верно (что то не так/сервер сломался/клиентская ошибка – запрос на несуществующий url), в консоль выведется сообщение об ошибке.

НО! чаще всего при работе с XMLHttpRequest используют событие load, оно проще потому что срабатывает только один раз, когда запрос уже полностью готов. НО! готов не значит, что завершён успешно. Т.к. load не отслеживает readyState, условие с ним будет выглядеть как:

if (request.status === 200)

На основании такого взаимодействия можно получать данные из баз данных, формировать на их основе списки чего-либо. Обрабатывать данные и выводить нужный результат, стоить по ним графики и т.д.

053 Реализация скрипта отправки данных на сервер (Практика 13)

Работаем на локальном сервере!

Задача: взять формы обратной связи которые есть в вёрстке (2 шт) собрать из них данные и отправить на сервер.

Для того, чтобы видеть, что бэкенд нормально принимает данные в корне проекта создаётся файл server.php с содержимым:

<?php

echo var\_dump($\_POST);

Команда берёт данные, которые пришли с клиента, преобразовывает их в строку и показывает обратно на клиенте – response от сервера с предыдущего урока.

Решение: т.к. формы две, функционал по отправке должен будет повторяться. Чтобы вручную не создавать два одинаковых обработчика, он будет обёрнут в функцию, которая будет вызываться при отправке формы (распространённая практика!). Будет использован объект XMLHttpRequest, данные будут отправляться в двух разных форматах – объект FormData (нормально используется в PHP) и объект JSON (в PHP требует декодирования). Код:

217:     const forms = document.querySelectorAll('form');

218:     const message = {

219:         loading: 'Загрузка',

220:         success: 'Спасибо! Мы с вами свяжемся',

221:         failure: 'Что-то пошло не так...'

222:     };

223:

224:     forms.forEach(item => {

225:         postData(item);

226:     });

227:

228:     function postData(form) {

229:         form.addEventListener('submit', (e) => {

230:             e.preventDefault();

231:

232:             let statusMessage = document.createElement('div');

233:             statusMessage.classList.add('status');

234:             statusMessage.textContent = message.loading;

235:             form.append(statusMessage);

236:

237:             const request = new XMLHttpRequest();

238:             request.open('POST', 'server.php');

239:

240:             // request.setRequestHeader('Content-type', 'multipart/form-data');

241:             const formData = new FormData(form);

242:

243:             request.send(formData);

244:

245:             request.addEventListener('load', () => {

246:                 if (request.status === 200) {

247:                     console.log(request.response);

248:                     statusMessage.textContent = message.success;

249:                 } else {

250:                     statusMessage.textContent = message.failure;

251:                 }

252:             });

253:         });

254:     }

В переменную forms получаем все формы по тэгу <form>. Прописывается функция postData() отвечающая за постинг данных, в качестве аргумента передаётся form. На форму form навешивается обработчик события submit срабатывающий при клике по кнопке в форме (у кнопок в вёрстке всегда по умолчанию type=”submit”) или нажатию enter и функция-обработчик с аргументов в виде объекта события е. Для отмены стандартного поведения браузера, прописывается e.preventDefault() такая команда в AJAX запросах должна быть в самом начале. В переменную request помещается объект XMLHttpRequest. Производится настройка запроса методом .open('POST', 'server.php') в качестве метода общения с сервером указывается POST и url на который будет отправляться запрос server.php. Далее будет использован объект FormData – самый простой способ подготовить данные для отправки из формы на сервер. Данные на сервер не всегда передаются в формате JSON! Всё зависит от сервера.

FormData – специальный объект позволяющий из определённой формы быстро сформировать данные заполненные пользователем, формат – ключ: значение. ВАЖНО: при использовании FormData, в вёрстке формы у всех элементов, данные из которых будут отправляться на сервер обязательно должен быть прописан атрибут name=”…” иначе FormData не сможет найти этот инпут и взять из него value, чтобы правильно сформировать объект!

Согласно документации FormData для его правильной работы необходимо установить заголовок запроса в виде request.setRequestHeader('Content-type', 'multipart/form-data');. НО! когда используется связка XMLHttpRequest + FormData заголовок устанавливать ненужно, он устанавливается автоматически!

Далее с помощью метода .send() содержимое переменной formData в виде объекта FormData содержащего данные сформированные на основании заполненной пользователем формы отправляется на сервер. После на request навешивается обработчик события по событию load. Функция-обработчик проверяет, если статус запроса 200, в консоль выведется ответ сервера request.response. А так же для информирования пользователя о ходе отправки запроса будут выводится текстовые сообщения, для этого создаётся новый объект message, содержащий список фраз для разных ситуаций. loading: 'Загрузка' – когда запрос ещё не ушёл, не успешен и не сломался. success: 'Спасибо! Мы с вами свяжемся' – когда всё ок, статус 200. failure: 'Что-то пошло не так...'. Далее динамически будет создаваться ещё один блок внутри <form>, для этого с помощью метода document.createElement('div') создаётся <div> и помещается в переменную statusMessage. Затем с помощью .classList.add('status') этому <div>’у добавляется класс status(если он заранее прописан в css). После, с помощью свойства .textContent = message.loading в <div> будет помещаться сообщение находящееся в свойстве .loading объекта message. Теперь важно отобразить этот элемент на странице, т.к. до этого он существует только в JS, для этого используется метод .append() - form.append(status.message). После этого statusMessage начинает существовать на странице и становится DOM узлом, следовательно его можно модифицировать при правдивом срабатывании условия (.status=200) statusMessage.textContent = message.success. Иначе в statusMessage будет выведено другое сообщение statusMessage.textContent = message.failure.

Для того чтобы под каждую форму «подвязать» функцию postData используется метод .forEach() с аргументом item. Внутри функция будет вызывать функцию postData для каждой формы postData(item).

Вместо во время и после отправки формы, вместо объекта message можно подставить блоки анимации загрузки и завершения, progress bar или другой контент.

Доработка скрипта: форма должна очищаться после отправки запроса, сообщения о прогрессе отправки должны пропадать через несколько секунд.

Для очистки формы, в случае успешной отправки (.status=200), прописывается метод .reset() – сбрасывание формы, альтернативный вариант – взять инпуты из формы, перебрать их и очистить их value.

Для удаления блока с сообщениями о прогрессе отправки, в случае успешной отправки устанавливается setTimeout с задержкой 2 секунды, с callback функцией которая будет вызывать метод remove() для элемента в переменной statusMessage - statusMessage.remove().

 request.addEventListener('load', () => {

                if (request.status === 200) {

                    console.log(request.response);

                    statusMessage.textContent = message.success;

                    form.reset();

                    setTimeout(() => {

                        statusMessage.remove();

                    }, 2000);

                } else {

                    statusMessage.textContent = message.failure;

                }

            });

Если сервер должен принимать данные в формате JSON:

240:             request.setRequestHeader('Content-type', 'application/json');

241:

242:             const formData = new FormData(form);

243:             const object = {};

244:             formData.forEach(function(key, value) {

245:                 object[key] = value;

246:             });

247:

248:             const json = JSON.stringify(object);

249:

250:             request.send(json);

Для JSON нужен заголовок ‘application/json’ кодировка указывается по желанию. Далее объект FormData необходимо превратить в формат JSON. Применяется следующий приём: создаётся пустой объект object; содержимое переменной formData перебирается методом .forEach() с callback функцией с аргументами key и value, соответствующие пары значений помещаются в object - object[key] = value; таким образом object становится обычным объектом а не формата FormData и его уже можно конвертировать в JSON. Используется метод JSON.stringify(object) для превращения object в объект JSON, результат записывается в переменную json. Затем запрос с содержимым переменной json отправляется на сервер request.send(json).

Нюанс связанный с бэкенд разработкой – php нативно не умеет работать с форматом данных JSON, чаще всего такие данные отправляются на сервер с использование, к примеру Node.js. Но поработать с таким типом данных можно, для этого нужно дополнить server.php:

<?php

$\_POST = json\_decode(file\_get\_contents("php://input"), true);

echo var\_dump($\_POST);

Всё что приходит от клиента, будет декодироваться функцией json\_decode, внутрь помещается указанная выше конструкция.

054 Красивое оповещение пользователя (Практика 14)

Задача: при отправке сообщения, пользователь будет видеть «спинер» сигнализирующий о том, что идёт загрузка, когда запрос полностью отправлен, будет показано видоизменённое модальное окно. Отправка формы>спинер>окно с благодарностью или с сообщением об ошибке.

Решение:

В основе окна с благодарностью будет уже существующее модальное окно, будет заменяться <div class="modal\_\_dialog">. В коде проверяющем успешную отправку, нужно скрыть существующий modal\_\_dialog и показать другой, затем передать в него текст сообщающий о статусе отправки.

Код:

function showThanksModal(message) {

        const prevModalDialog = document.querySelector('.modal\_\_dialog');

        prevModalDialog.classList.add('hide');

        openModal();

        const thanksModal = document.createElement('div');

        thanksModal.classList.add('modal\_\_dialog');

        thanksModal.innerHTML = `

        <div class="modal\_\_content">

            <div class="modal\_\_close" date-close>×</div>

            <div class="modal\_\_title">${message}</div>

        </div>

        `;

        document.querySelector('.modal').append(thanksModal);

        setTimeout(() => {

            thanksModal.remove();

            prevModalDialog.classList.add('show');

            prevModalDialog.classList.add('hide');

            closeModal();

        }, 4000);

    }

Объявляется функция showThanksModal с аргументом message, которая будет отображать модальное окно с сообщением из одноимённого объекта message. В переменную prevModalDialog помещается <div> с классом "modal\_\_dialog" со страницы. С помощью метода .classList.add('hide') и добавляется класс hide и предыдущий контент скрывается, НЕ! Удаляется, это нужно для того, чтобы пользователь при желании мог повторно открыть модальное окно и отправить форму, если удалить блок, то такой функционал не сработает. Затем вызывается описанная ранее функция openModal() открывающая модальное окно. Начинается создание нового контента. В переменную thanksModal помещается созданный методом .createElement() элемент <div>. Методом .classList.add() ему добавляется класс "modal\_\_dialog" (один «заполненный» <div> с классом "modal\_\_dialog" заменяется таким же «пустым»). Далее с помощью свойства .innerHTML внутрь элемента в переменной thanksModal помещается вёрстка модального окна с «крестиком» (спецсимвол из html, реализация работы ниже) и сообщением message(которое берётся из одноимённого объекта). С помощью .querySelector('.modal') модальное окно получается со страницы, в переменную не сохраняется, т.к. нигде далее использоваться не будет. Далее методом .append(thanksModal) в него помещается содержимое thanksModal.

Если пользователь опять захочет открыть модальное окно и отправить форму, через определённое время всё должно возвращаться на свои места. Для этого используется setTimeout с задержкой в 4000 миллисекунд. Внутри callback функции в setTimeout будет происходить удаление thanksModal методом .remove() (если пользователь повторно отправит форму, thanksModal создастся заново) – шаг обязательный к исполнению, чтобы не накапливались блоки!. Далее нужно показать предыдущий контент, реализуется методом добавления класса show к prevModalDialog и удалением класса hide. Т.к. пользователь всё это время смотри на модальное окно, чтобы не смущать его, лучше закрыть окно вызовом функции closeModal().

Решение вопроса с динамически созданным «крестиком»

const modalTrigger = document.querySelectorAll('[data-modal]'),

        modal = document.querySelector('.modal');

…

modal.addEventListener('click', (e) => {

        if (e.target === modal || e.target.getAttribute('data-close') == '') {

            closeModal();

        }

    });

Динамически созданный «крестик» не будет реагировать на те действия, которые были повешены на него изначально, для решения этой проблемы будет использовано делегирование событий. Для этого нужно удалить переменную modalCloseBtn и навешенный на неё обработчик событий, т.к. они не будут больше использоваться, взаимодействие будет происходить с родителем ранее полученным в переменную modal. Модифицируется условие в callback функции обработчика событий навешенного на modal до вида e.target === modal || e.target.getAttribute('data-close') == '' Если цель клика подложка modal или у цели клика присутствует дата атрибут data-close (условие имеет вид ('data-close') == '' потому что мы в него ничего не помещаем, он просто есть или нет), то будет вызвана функция closeModal() закрывающая модальное окно.

Возвращаемся к отправке. statusMessage.remove() освобождается setTimeout, т.к. statusMessage будет использоваться только для loading, т.е. для отображения «спинера» на странице. В условии if, строка statusMessage.textContent = message.success замеряется вызовом функции showThanksModal с тем же сообщением, что и ранее message.success. Таким же образом модифицируется условие else - showThanksModal(message.failure). Вместо текста внутрь message.loading подставляется путь к картинке «спинера», код переменной statusMessage переписывается под эту картинку – вместо <div> создаётся <img>, строка с добавлением несуществующего класса удаляется. Для statusMessage внутри которого находится <img> прописывается атрибут src которому присваивается значение message.loading. (можно использовать .setAttribute('src', message.loading)). Далее удаляется строка statusMessage.textContent = message.loading; т.к. message.loading уже содержит адрес картинки а не текст, вместо неё с помощью свойств .style.cssText прописываются инлайн стили для statusMessage.

Т.к. нижняя форма свёрстана при помощи флексов, при добавлении «спинера» её элементы начинают наезжать друг на друга. Для того, чтобы она не «глючила», вместо метода form.append(statusMessage) прописывается метод form.insertAdjacentElement('afterend', statusMessage), который будет вставлять «спинер» под/после формы – afterend, а не в неё. На верхней форме/в модальном окне это не отразится, т.к. там «спинер» и так отображался снизу.

const message = {

        loading: 'img/form/spinner.svg',

        success: 'Спасибо! Мы с вами свяжемся',

        failure: 'Что-то пошло не так...'

    };

…

 function postData(form) {

        form.addEventListener('submit', (e) => {

            e.preventDefault();

            let statusMessage = document.createElement('img');

            statusMessage.setAttribute('src', message.loading);

            statusMessage.style.cssText = `

                display: block;

                margin: 0 auto;

            `;

            form.insertAdjacentElement('afterend', statusMessage);

            const request = new XMLHttpRequest();

            request.open('POST', 'server.php');

            request.setRequestHeader('Content-type', 'application/json');

            const formData = new FormData(form);

            const object = {};

            formData.forEach(function(key, value) {

                object[key] = value;

            });

            const json = JSON.stringify(object);

            request.send(json);

            request.addEventListener('load', () => {

                if (request.status === 200) {

                    console.log(request.response);

                    showThanksModal(message.success);

                    form.reset();

                    statusMessage.remove();

                } else {

                    showThanksModal(message.failure);

                }

            });

        });

    }

055 Promise(ES6)

Не поддерживаются в IE11, для реализации промисов в нём, нужно использовать трансплитер Babel=>Polyfill он будет превращать код с промисами в старый.

Если произошло что-то, то мы обещаем/promise, что выполнится какое-то действие.

Часто можно встретить много действий которые должны выполняться друг за другом и только при условии выполнении предыдущего(много вложенных друг в друга и передаваемых в виде аргумента callback функций) - “callback hell”.

? – тернарный(условный) оператор, синтаксис: условие ? выражение1 : выражение2 часто используется в качестве укороченного варианта условного оператора if. Тернарный потому что в его работе участвуют 3 аргумента. Единственный тернарный оператор в JS на данный момент.

let drink = 0;

function shoot(arrow, headshot, fail) {

    console.log('Вы сделали выстрел ...');

    setTimeout(function() {

        Math.random() > 0.5 ? headshot({}) : fail('Вы промахнулись');

    }, 3000);

}

function win() {

    console.log('Вы победили!');

    (drink == 1) ? buyBeer(): giveMoney();

}

function buyBeer() {

    console.log('Вам купили пиво');

}

function giveMoney() {

    console.log('Вам заплатили!');

}

function lose() {

    console.log('Вы проиграли!');

}

shoot({},

    function(mark) {

        console.log('Вы попали в цель!');

        win(mark, buyBeer, giveMoney); // Тут может быть овердофига функций

    },

    function(miss) {

        console.error(miss);

        lose();

    }

);

Пример callback hell.

Переработка кода с помощью promise, часть I «как создать промис»:

let drink = 0;

function shoot(arrow) {

    console.log('Вы сделали выстрел ...');

    let promise = new Promise(function(resolve, reject) {

        setTimeout(function() {

            Math.random() > 0.5 ? resolve({}) : reject('Вы промахнулись');

        }, 3000);

    });

    return promise;

}

Выполняется функция shoot – сообщение выводится в консоль, создаётся новое обещание(promise), которое содержит в себе 2 аргумента/состояния: resolve – когда обещание выполнится и reject – когда нет. Происходит расчёт рандомного выстрела и вызов соответствующей функции через 3 сек. Promise возвращается.

Часть II «как использовать промис»:

shoot({}).then(mark => console.log('Вы попали в цель!')).then(win).catch(lose);

Для того, чтобы описать цепочку событий, при состоянии resolve используется оператор/метод then(), для описания событий при состоянии reject используется метод .catch(). Набор промежуточных функций тот же, что и в примере callback hell.

Если после .catch() прописан .then(), он будет выполнен всегда не зависимо от того, что вернётся от промиса resolve или reject.

Основная суть промисов – взаимодействие с сервером и использование в AJAX запросах, таким образом код делается асинхронным и более отзывчивым к пользователю.

Пример из обновлённого курса:

Код имитирует работу с сервером – запрос данных, их подготовку, модификацию и возвращение объекта:

console.log('Запрос данных...');

setTimeout(() => {

    console.log('Подготовка данных ...');

    const product = {

        name: 'TV',

        price: 2000

    };

    setTimeout(() => {

        product.status = 'ordered';

        console.log(product);

    }, 2000);

}, 2000);

Вместо setTimeout могут быть другие асинхронные операции, например работа с сервером. Существуют скрипты в которых не 2 действия, а 10-15 действий подряд – это колбеки внутри других колбеков, тогда структура кода сильно разрастается и поддерживать такой код сложно.

Далее будут выполнены те же действия, но с помощью промисов.

Для создания промиса, объявляется новая переменная p в которую помещается новый экземпляр глобального класса Promise, в конструктор этого класса необходимо передать callback функцию , которая принимает два аргумента resolve и reject, которые так же являются функциями (вместо них могут подставляться реальные функции!). Имеем callback функцию в конструкторе класса Promise, внутри которой как правило, прописывается асинхронный код.

const p = new Promise(function(resolve, reject) { //асинхронный код });

Функция resolve() вызывается тогда, когда асинхронная операция закончена успешно. Вызов resolve() говорит о том, что promise завершил своё выполнение:

const p = new Promise(function(resolve, reject) {

    setTimeout(() => {

        console.log('Подготовка данных ...');

        const product = {

            name: 'TV',

            price: 2000

        };

        resolve();

    }, 2000);

});

У промисов есть методы .then() .catch() .finnaly() в качестве аргументов принимающие callback функции. Далее прописывается метод .then(), читается как «потом/после», подразумевается, что его содержимое выполнится при успешном выполнении операции внутри промиса. = вызов функции resolve(). Успешное выполнение содержимого promise => выполнение resolve() => выполнение p.then():

console.log('Запрос данных...');

const p = new Promise(function(resolve, reject) {

    setTimeout(() => {

        console.log('Подготовка данных ...');

        const product = {

            name: 'TV',

            price: 2000

        };

        resolve(product);

    }, 2000);

});

p.then(product => {

    setTimeout(() => {

        product.status = 'ordered';

        console.log(product);

    }, 2000);

});

Для того чтобы передать объект product из одной функции в другую их необходимо «вернуть» из функции, это можно сделать с помощью функции resolve(product) , тогда функция resolve( фактически = p.then() ) будет принимать в качестве аргумента product, возвращенный из предыдущего этапа p.then(product... Внутри p.then() снова асинхронная операция, модифицирующая product и выводящая его в консоль.

Получается тот же результат, что и при работе с callback функциями.

Если нужно сделать ещё какие-то действия, нужно обернуть нижний setTimeout в промис p2, соответственно вызвать resolve() и прописать .then() чтобы эти действия выполнились:

console.log('Запрос данных...');

const p = new Promise(function(resolve, reject) {

    setTimeout(() => {

        console.log('Подготовка данных ...');

        const product = {

            name: 'TV',

            price: 2000

        };

        resolve(product);

    }, 2000);

});

p.then(product => {

    const p2 = new Promise((resolve, reject) => {

        setTimeout(() => {

            product.status = 'ordered';

            resolve(product);

        }, 2000);

    });

    p2.then(productMod => {

        console.log(productMod);

    });

});

Чтобы не делать такую вложенность, которая почти совпадает с тем, что можно получить при помощи колбэков, можно воспользоваться ключевой особенностью промисов, а именно внутри метода .then() вместо создания ещё одного промиса и обращения к этому новому промису p2.then(… можно вместо создания переменной вернуть новый промис:

console.log('Запрос данных...');

const p = new Promise(function(resolve, reject) {

    setTimeout(() => {

        console.log('Подготовка данных ...');

        const product = {

            name: 'TV',

            price: 2000

        };

        resolve(product);

    }, 2000);

});

p.then(product => {

    return new Promise((resolve, reject) => {

        setTimeout(() => {

            product.status = 'ordered';

            resolve(product);

        }, 2000);

    });

}).then(productMod => {

    console.log(productMod);

});

В первом .then() получаем данные (объект product), далее возвращаем новый промис, и т.к. возвращаем промис, можем вызвать метод .then(productMod… с новыми данными, ранее переданными функции resolve. Приложение работает так же.

Преимущества:

Получается один уровень вложенности, читается лучше чем колбэки внутри колбэков. Объект p можно передавать куда угодно – между модулями(если есть система модулей) или функциями и с помощью .then() говорить что нужно сделать когда промис будет выполнен. С помощью метода .then() можно «чейнить»/chein , т.е. записывать цепочку операций, выполняющихся одна за другой, в ходе которых можно модифицировать и возвращать/передавать данные, прописывается как несколько .then в столбик.

Пример выполнения кода по цепочке и возвращения данных:

console.log('Запрос данных...');

const p = new Promise(function(resolve, reject) {

    setTimeout(() => {

        console.log('Подготовка данных ...');

        const product = {

            name: 'TV',

            price: 2000

        };

        resolve(product);

    }, 2000);

});

p.then(product => {

    return new Promise((resolve, reject) => {

        setTimeout(() => {

            product.status = 'ordered';

            resolve(product);

        }, 2000);

    });

}).then(productMod => {

    productMod.modify = true;

    return productMod;

}).then(productMod => {

    console.log(productMod);

})

Из второго .then() возвращается модифицированный объект и выводится в консоль в третьем .then(). Происходит последовательное выполнение кода, не смотря на то, какой код внутри синхронный или асинхронный.

Ещё одно преимущество промисов: если используются колбэки, приходится добавлять if’ы проверки статуса (status === 404, не показывать данные, а выводить ошибку), на этот случай у promise есть метод функция reject и метод .catch(), который обычно ставится в конце цепочки. Эта функция и метод специально предназначены для работы с ошибками выполнения операций. Для того, чтобы .catch() выполнился, в промисе должен быть вызов reject:

console.log('Запрос данных...');

const p = new Promise(function(resolve, reject) {

    setTimeout(() => {

        console.log('Подготовка данных ...');

        const product = {

            name: 'TV',

            price: 2000

        };

        resolve(product);

    }, 2000);

});

p.then(product => {

    return new Promise((resolve, reject) => {

        setTimeout(() => {

            product.status = 'ordered';

            console.log(product);

            reject();

        }, 2000);

    });

}).then(productMod => {

    productMod.modify = true;

    return productMod;

}).then(productMod => {

    console.log(productMod);

}).catch(() => {

    console.error('Произошла ошибка');

});

Если что-то пошло не так, после вызова reject, сработает участок кода с .catch(), цепочка методов .then() будет пропущена.

Метод .finnaly() – это метод вызываемый при любом исходе, не зависимо от того была ли ошибка или всё прошло успешно. Обычно прописывается в САМОМ конце. Прописывается так же, как и вышеописанные методы:

}).finally(() => {

    console.log('Finally');

});

Пример использования: очистка формы от данных.

Дополнительные возможности и методы .all() и .race(), которые упрощают работу с асинхронностью:

const test = time => {

    return new Promise(resolve => {

        setTimeout(() => resolve(), time);

    });

};

Функция test с параметром time возвращающая промис(можно передавать только одну функцию), которая вызывает resolve() через время time, которое мы передаём. Использование функции test:

test(1000).then(() => console.log('1000 ms'));

test(2000).then(() => console.log('2000 ms'));

Обращаемся к функции, указываем задержку, когда задержка пройдёт, в консоль будет выведено сообщение. Запись выглядит лаконично и удобно, относительно использования колбэков.

Метод .all() принимает массив промисов и возвращает промис(у которого есть все методы) который выполнится только тогда, когда выполнятся все промисы в указанном массиве:

Promise.all([test(2000), test(5000)]).then(() => {

    console.log('All promises done');

});

Пример использования : отправляются несколько запросов на разные сервера, для получения разных картинок, сервера отвечают с разным промежутком по времени, благодаря методу .all() изображения появятся на странице только когда все они будут загружены.

Метод .race() принимает те же аргументы, что и .all() и возвращает промис, который срабатывает тогда, когда выполнится первый из переданных промисов:

Promise.race([test(2000), test(5000)]).then(() => {

    console.log('First promise done');

});

056 Fetch API (Практика 15)

API расшифровывается как Application Programming Interface интерфейс программирования приложения, по простому – это набор данных и возможностей, которые предоставляет какое-то готовое решение. Ранее мы уже работали с DOM API – набор методов позволяющих работать с элементами на странице. Бывают Google Maps API, Yandex Maps API и т.д. – т.е. разработчику предоставляются готовые методы и свойства, которые можно использовать.

Fetch API уже встроен в браузер, позволяет общаться с сервером и построен на промисах. Для использования Fetch API достаточно прописать fetch('https://jsonplaceholder.typicode.com/todos/1') внутри круглых скобок в кавычках указывается url адрес сервера куда посылается запрос. Если больше ничего не указывать, это будет классический GET запрос, который просто получит данные из этого url. Главное, что fetch() возвращает promise, который можно обработать при помощи цепочки .then.

Пример №1: отправляем GET запрос, получаем данные и как-то их обрабатываем. В качестве url сервера, куда будут посылаться запросы(чтобы не создавать у себя на сервере JSON файл куда будут отправляться запросы) будет использовано готовое решение <https://jsonplaceholder.typicode.com> – небольшая база данных в формате JSON, лежащая в интернете к которой можно обращаться для тестирования своих ресурсов.

fetch('https://jsonplaceholder.typicode.com/todos/1')

        .then(response => response.json())

        .then(json => console.log(json));

Первая строка отправляет запрос. Во второй строке используется встроенная конструкция fetch() - ответ, который вернулся в формате JSON через => конвертируется в обычный JS объект методом .json(). response.json() работает сродни JSON.parse() (разбирает строку JSON, возможно с преобразованием получаемого в процессе разбора значения). Важно! response.json() возвращает promise и тело ответа от сервера(непосредственно данные), для того, чтобы можно было продолжить цепочку .then. В третьей строке уже используется обычный JS объект, который выводится в консоль.

У fetch() есть ещё несколько методов – вместо .json() можно использовать .text() и несколько других редко используемых методов для того чтобы модифицировать response приходящий от сервера, список методов можно увидеть при выводе response в консоль.

Пример№2: отправить другие запросы POST, PUT при помощи fetch().

Для этого в fetch() через запятую добавляется объект с настройками:

fetch('https://jsonplaceholder.typicode.com/posts', {

            method: 'POST',

            body: JSON.stringify({ name: 'Alex' }),

            headers: {

                'Content-type': 'application/json'

            }

        })

Свойств может быть много, но обязательными свойствами являются method и body – тело, которое будет отправляться. Желательно так же указывать заголовки headers (указываются в виде объекта!). В ответ в консоль выводится {name: "Alex", id: 101} – на сервере было 100 постов, эта запись стала 101 (id: 101), в данном примере это значит что всё отработало правильно.

Fetch запросы удобнее XMLHttpRequest – url задаётся одной строкой, все нужные настройки идут объектом, поэтому данная технология сейчас востребована.

Практика 15:

Переписываем функционал проекта с использованием fetch(). Т.к. всё зависит от сервера, который будет использован, будет рассмотрено два варианта:

1. Отправление классической FormData.
2. Отправление файла JSON

Отправляем данные в формате FormData:

Удаляем строки создание объекта XMLHttpRequest, вместо него будет fetch():

237 const request = new XMLHttpRequest();

238 request.open('POST', 'server.php');

Информация из настройки заголовка методом .setRequestHeader(), перемещается в headers внутри fetch(), строка удаляется:

244 request.setRequestHeader('Content-type', 'application/json');

Вместо строки:

request.send(json);

Вставляется fetch():

fetch('server.php', {

                    method: 'POST',

                    // headers: {

                    //     'Content-type': 'application/json'

                    // },

                    body: formData

      })

Код с объектом headers временно комментируется (работаем с formData).

Т.к. сейчас будет отправляться formData, комментируется следующий участок кода преобразовывающий её в JSON:

239:             // const object = {};

240:             // formData.forEach(function(value, key) {

241:             //     object[key] = value;

242:             // });

243:

244:             // const json = JSON.stringify(object);

В server.php комментируется вторая строка, отвечающая за работу с JSON:

// $\_POST = json\_decode(file\_get\_contents("php://input"), true);

Комментируется код обработки результатов запроса, он будет переписан ниже с помощью промисов:

254:             // request.addEventListener('load', () => {

255:             //     if (request.status === 200) {

256:             //         console.log(request.response);

257:             //         showThanksModal(message.success);

258:             //         form.reset();

259:             //         statusMessage.remove();

260:             //     } else {

261:             //         showThanksModal(message.failure);

262:             //     }

263:             // });

Код обработки результатов запроса переписывается с использованием методов промисов:

.then(data => {

console.log(data);

showThanksModal(message.success);

      statusMessage.remove();

})

.catch(() => {

showThanksModal(message.failure);

})

.finally(() => {

form.reset();

});

Вместо request.response подставляется data – те данные, что возвращаются из промиса=которые вернул сервер. form.reset(); перемещается в .finally() т.к. очистка формы должна производиться в любом случае. Логика работы: действия, выполняемые когда всё ОК (if=true) помещаются в .then(), когда что-то не так (if=false/else) помещаются в .catch(), действия выполняемые не зависимо от результата помещаются в .finally().

Код, используемый для обработки результатов с использованием XMLHttpRequest (строка 254-263) удаляется за ненадобностью.

Пока получаем в консоли Response {type: "basic", url: "http://project3/server.php", redirected: false, status: 200, ok: true, …}.

Чтобы чётко понимать что уходит на сервер, ответ сервера можно модифицировать:

.then(data => data.text())

Тогда в консоли получаем:

array(2) {

[“name”]=>

string(6) “gfgfgf”

[“phone”]=>

string(9) “436534535”

}

Это те же данные, что были введены. В консоли разработчика проверяем вкладку Network>server.php> внутри есть объект FormData, содержащий name: gfgfgf phone: 436534535, значит всё ок.

Отправляем данные в формате JSON:

Раскомментируем код трансформирующий FormData в JSON (аргументы функции value, key должны быть именно в таком порядке, иначе они меняются местами хз почему):

const object = {};

formData.forEach(function(value, key) {

object[key] = value;

});

const json = JSON.stringify(object);

Далее объект json передаётся в feth() в body, чтобы не создавать переменную, можно подставить сразу JSON.stringify(object) и раскомментировать body:

 fetch('server.php', {

        method: 'POST',

        headers: {

                 'Content-type': 'application/json'

                    },

                    body: JSON.stringify(object)

                })

В server.php раскомментируется вторая строка, отвечающая за работу с JSON:

$\_POST = json\_decode(file\_get\_contents("php://input"), true);

При тестировании .catch() не возникает сообщения о том, что что-то не так, и в консоль выводится большое сообщение об ошибке. Особенность работы fetch() – промис, который запускается при помощи fetch() не перейдёт в состояние отклонено/rejected из-за ответа http(связанного с http протоколом), который считается ошибкой (404, 500, 501, 502…), он всё равно выполнится нормально, всё что у него поменяется – это свойство status, которое перейдёт в состояние false. Т.е. для fetch() главное, что он смог выполнить этот запрос, это приравнивается к resolve, reject же может возникнуть только при сбое сети или если что-то помешало запросу выполниться. Выходит, что блок кода .catch() отрабатывает правильно, но его работу можно проверить при Network>Offline.

057 Методы перебора массивов

.forEach() никогда не возвращает новый массив, а перебирает текущий.

Методы возвращающие новый массив:

1) Метод .filter() фильтрует элементы внутри массива, вовнутрь принимает callback функцию, которая перебирает элементы и возвращает только те, которые подходят под условия. Задача: получить имена длинной менее 5 символов. Код:

const names = ['Ivan', 'Alex', 'Ksenia', 'Volandemort'];

const shortNames = names.filter(function(name) {

    return name.length < 5;

});

[ 'Ivan', 'Alex' ]

name – это элемент массива names, т.к. нужно возвращать элементы, напрямую прописывается return, условие возврата можно записать так как в примере или через if (условие зависит от задачи).

2) Метод .map() позволяет взять исходный массив и изменить каждый элемент внутри него, в итоге получается новый массив с изменёнными данными. Принимает callback функцию перебирающую, преобразующую и возвращающую элементы массива. Задача: привести содержимое массива к нижнему регистру. Код:

const answers = ['IvAn', 'AnnA', 'HanNah'];

const result = answers.map(item => item.toLowerCase());

[ 'ivan', 'anna', 'hannah' ]

Применён синтаксис стрелочной функции, item – элемент массива, фигурных скобок {} нет и return явно не прописан т.к. действие только одно.

Отдельную переменную для нового массива создавать не обязательно, можно сохранить новый массив в ранее созданной переменной, предварительно изменив её тип на let. Считается неоднозначной практикой.

3) Метод .some() проверяет исходный массив, если хоть один элемент подходит под условие, метод вернёт true, если нет – false. Внутрь принимает callback функцию. Задача: проверить есть ли в массиве числа. Код:

const arr = ['4', 'sddfg', 'rerter'];

console.log(arr.some(item => typeof(item) === 'number'));

false

В консоли false, потому что ‘4’ – это строка.

4) Метод .every() проверяет все ли элементы внутри массива подходят под условие, да – вернёт true, нет -–вернёт false. Внутрь принимает callback функцию. Задача: проверить все ли элементы являются строками. Код:

const arr = ['4', 'sddfg', 'rerter'];

console.log(arr.every(item => typeof(item) === 'string'));

true

5) Метод .reduce() перебирает и схлопывает/собирает массив в единое целое. Внутрь принимает callback функцию.

Задача№1: получить сумму всех элементов внутри массива. Код:

const numbers = [4, 5, 1, 3, 2, 6];

const result = numbers.reduce((sum, current) => sum + current);

                            //  0       4

                            //  4       5 скорее всего метод начинает работу тут

                            //  9       1

                            //  10      3

                            //  13      2

                            //  15      6

                            //  21

21

Функция с двумя аргументами, первый – сумма всех элементов, которая изначально условно = 0 и перезаписывается с каждой итерацией, второй – элемент массива, который меняется при каждой итерации цикла перебора. Внутри callback функции можно производить не только сложение, но и другие математические операции. Смысл метода остаётся прежним – массив «схлопывается» до единственного значения.

Задача№2: преобразовать массив в строку, где элементы массива будут представлены через запятую. Код:

const strings = ['apple', 'carrot', 'cucumber'];

const result = strings.reduce((sum, current) => `${sum}, ${current}`);

console.log(result);

apple, carrot, cucumber

Перед apple нет запятой т.к. .reduce() скорее всего начинает сразу «схлопывать» элементы массива между собой, а не вначале с пустотой.

Метод .reduce() в качестве дополнительного аргумента для callback функции может принять начальное значение:

const result = numbers.reduce((sum, current) => sum + current, 3);

24

Практический пример:

Получаем объект:

const obj = {

    ivan: 'persone',

    anna: 'persone',

    dog: 'animal',

    cat: 'animal',

};

В объекте не может быть одинаковых названий свойств!

Задача: получить имена людей, находящиеся в этом объекте.

Будет использован метод .entries() – метод для работы с объектом позволяющий превратить его в матрицу/массив массивов:

const newArr = Object.entries(obj);

[

  [ 'ivan', 'persone' ],

  [ 'anna', 'persone' ],

  [ 'dog', 'animal' ],

  [ 'cat', 'animal' ]

]

Считается удобным методом, когда нужно поработать с объектом как с массивом и что-то с ним сделать.

Далее будет использован приём chain – когда операции записываются в цепочку. Логика решения – нужно получить два массива, у которых вторые значения равны 'persone', Для этого отфильтруем элементы массивов и оставим только те, у которых вторые элементы равны 'persone'. Результат будет сохранён в newArr.

const newArr = Object.entries(obj)

.filter(item => item[1] === 'persone');

[ [ 'ivan', 'persone' ], [ 'anna', 'persone' ] ]

Теперь из массивов нужно получить только первые элементы – имена и записать их в новый массив:

const newArr = Object.entries(obj)

.filter(item => item[1] === 'persone')

.map(item => item[0]);

[ 'ivan', 'anna' ]

Выводы: зная методы перебора массивов и как сделать их объекта массив, многие задачи можно решить в одну строку.

058 Подробно про npm и проект. JSON-server (Практика 16)

npm пакеты – «кусочки» готового кода лежащие на серверах, которые можно установить себе для дальнейшего использования.

Перед установкой npm пакетов, нужно объявить системе о том, что данный проект будет их содержать = превращается в npm проект.

Создание npm проекта:

-открыть терминал в VS Code

-прописать npm init, затем указать package name(если нужно изменить стандартное), version, description, entry point, test command – команды для тестов, git repository, keywords, author, license

-на вопрос Is this OK ? жмём y + enter или просто enter

-в проекте появляется файл package.json содержащий все настройки выше, а так же будет содержать информацию об установленных пакетах

Установка npm пакетов:

Первое слово - откуда: npm; второе - что делаем?: install или i; третье - что устанавливаем?: json-server; четвёртое – дополнительные настройки: «ничего»(пакет ставится локально, работает только внутри проекта) или -g (глобально, пакет работает в любой части системы); пятое – указать флаг как пакет влияет(для разработки/ для работы) на проект: --save-dev(т.н. «зависимость для разработки» - будет использован только при разработке) или –save (для работы, пример jQuery, React). Локальная установка является предпочтительной, тогда файл package.json будет конкретно говорить какие пакеты и их версии нужно установить для работы с проектом.

Установка JSON-server. Команда в консоли: npm i json-server --save-dev. Работа пакета схожа с <https://jsonplaceholder.typicode.com> позволяет обращаться на фейковый бэкенд и тестировать функционал. После установки, в файле package.json появилась зависимость:

"devDependencies": {

    "json-server": "^0.16.3"

  }

Если ставить пакет без “-dev” devDependencies будет без dev. Так же после установки появилась папка node-modules хранящая все npm пакеты установленные в проект и файл package-lock.json.

Папку node-modules лучше не трогать! Она занимает много места относительно проекта, поэтому в проекте всегда должен присутствовать файл .gitignore в котором прописаны файлы и папки, которые ненужно пушить, в том числе и node-modules. Папка удаляется специальной командой, вручную очень редко. Node-modules устанавливается так же одной командой при наличии файла package.json, команда: npm i. Файл package-lock.json так же лучше не трогать! Содержит различные пакеты, зависимости и пути к ним(мини-зависимости установленных пакетов).

Начало работа с JSON-server:

Простой сервер для работы с JSON файлами, которые можно использовать как маленькую базу данных. Поддерживает работу многих методов, в т.ч. ‘POST’. Эмулирует работу бэкенда.

Добавляем к проекту db.json – база данных содержащая материалы/данные для построения карточек товаров. В поле requests будут записываться обращения/заявки пользователя, когда он отправляет форму с сайта.

fetch('db.json')

        .then(data => data.json())

        .then(result => console.log(result));

Получаем доступ к базе данных с помощью fetch(), из него возвращается promise который можно обработать при помощи .then(). Первый .then преобразует ответ от сервера из формата JSON в формат обычного JS объекта, второй .then() выведет этот объект в консоль. В консоли будет объект с массивами с объектами внутри.

Запуск JSON-server, команда в консоли: npx json-server --watch db.json npx – если установлен локально ?. Получаем в консоли пути куда можно делать запросы:  
\{^\_^}/ hi!

Loading db.json

Done

Resources

http://localhost:3000/menu

http://localhost:3000/requests

Home

http://localhost:3000

Type s + enter at any time to create a snapshot of the database

Watching...

Для правильной работы с сервером, нужно подставить <http://localhost:3000/menu> в fetch():

fetch('http://localhost:3000/menu') //.then те же что и выше

В консоли получаем уже не объект а массив с объектами, эти данные будут в дальнейшем использованы на клиенте. В консоли VS Code появилось GET /menu 200 6.059 ms - - информация о том какие запросы ушли, с каким ответом и за сколько это произошло.

059 Получение данных с сервера. Async\_Await (ES8) (Практика 17)

Задача на урок: с помощью взаимодействия с сервером, построить карточки на странице. Познакомиться с операторами Async\_Await. Улучшить поведение отправки данных с форм на сервер.

Данные для «карточек» чаще всего не забиты внутри JS, а приходят с сервера, т.к. цены, ассортимент и продукты могут часто меняться, администратор меняет их через административную панель(Wordpress – готовое решение или самописная), а клиент(сайт) просто отображает изменённые данные – это удобно.

Оптимизация функций:

Пока к каждой форме подвязывается одинаковое действие, но в реальности они могут отличаться, например отправлять данные на разные url или ответы от сервера могу приходить в разных форматах=разные заголовки и т.д. Для этого функционал по общению с сервером выносят в отдельные функции.

const postData = (url, data) => {

        const result = fetch(url, {

            method: 'POST',

            headers: {

                'Content-type': 'application/json'

            },

            body: data

        });

        return result.json();

    };

Создаётся переменная postData. В неё будет помещён функционал отвечающий за постинг данных/отправку на сервер. Старую функцию с таким названием переименовываем в bindPostData – отвечает за привязку постинга, меняем название везде, где она была использована (в forms.forEach…). Реализуем дополнительную «фичу» - когда происходит запрос, можно сразу на этом же этапе обработать те данные, которые пришли(всё ок/ не ок, отобразить сообщение для пользователя). Для этого внутри функционального выражения postData объявляется переменная result, в эту переменную будет сохраняться promise, который возвращается от fetch(). Далее fetch запрос настраивается для отправки POST запроса, в качестве первого аргумента прописывается url, далее внутрь копируется содержимое описанного ранее/ниже fetch(). Т.к. используется сервер JSON, данные будут поститься в формате JSON, следовательно оставляем headers: { 'Content-type': 'application/json'}, но в будущем в функцию postData можно будет передавать аргументы влияющие на заголовки(для работы с другим типом сервера). Настраиваем аргументы функции, которые будут приходить: url, который передаётся дальше в fetch() и data – данные, которые будут поститься в этой функции, data будет передаваться в fetch(… body: data) как тело запроса. Функция postData будет возвращать result декодированный методом .json(), т.е. promise с телом ответа от сервера в формате объекта. Зачем возвращать ? Потому что возвращается promise, который можно в дальнейшем обработать через цепочку .then(). В итоге функция postData занимается тем, что настраивает запрос, посылает запрос на сервер с помощью fetch(), получает ответ (к примеру, что запостили успешно) и трансформирует ответ выводя соответствующие сообщения, очищая форму и т.д. как в уроке 056 Fetch API (Практика 15).

Почему нельзя вызвать функцию postData вместо описанного ранее/ниже fetch() ведь они выполняют одинаковые действия ? Потому что внутри postData АСИНХРОННЫЙ код – он не ждёт выполнения какого-либо другого кода. Разбираемся: при запуске postData, в первой строке мы начинаем делать запрос, т.е. сначала отрабатывает код справа - происходит запрос (когда придёт ответ от сервера неизвестно), далее объявляется переменная result и пока нет ответа, в неё помещается ничего, далее происходит ошибка, т.к. в строке return result.json(); метод .json() вызывается у «обещания ничего» и такой метод у него в данном случае небудет.

Решение: написать механизм превращения асинхронного кода в синхронный или его подобие, чтобы дождаться результатов отправки запроса, сделать это нужно в двух местах, т.к. и fetch() и return result.json() возвращают promise и нужно дождаться результата этого промиса, прежде чем его возвращать.

Для решения такой проблемы появились операторы Async\_Await, всегда используются в паре. async обычно ставится перед функцией (скобкой с параметрами), он означает, что перед внутри функции будет асинхронный код. Его парный оператор await ставится перед теми операциями, которые необходимо дождаться. Дополненный код:

const postData = async(url, data) => {

        const result = await fetch(url, {

            method: 'POST',

            headers: {

                'Content-type': 'application/json'

            },

            body: data

        });

        return await result.json();

    };

Принцип работы дополненной функции postData: формируется запрос методом fetch(), т.к. перед ним стоит await, остальной код будет дожидаться результатов(неважно каких) этого запроса. Тут асинхронный код начинает быть похожим на синхронный, но это не так, т.к. код идущий далее не блокируется, просто “JS” будет ждать окончания запроса вплоть до 30 секунд положенных по стандарту и только после возвращения результата, await пропустит работу кода дальше. В переменную result поместится результат полученный от сервера, следовательно у него появится метод .json(), который, в свою очередь возвращает promise. Скорость работы метода .json() зависит от размеров JSON объекта в переменной result, следовательно код так же асинхронный, следовательно так же нужен await.

Смысл работы с операторами Async\_Await – если предполагается, что в проекте будут какие-то таймауты или работа с сервером, нужно заранее продумывать нужную последовательность исполнения кода и при необходимости использовать эти операторы.

Теперь когда функция postData можно вызвать её с соответствующими аргументами вместо описанного ранее/ниже fetch():

postData('server.php', JSON.stringify(object))

аргументы: 'server.php' – url адрес сервера, JSON.stringify(object) – body/тело запроса, которое пойдёт на сервер.

Старый код метода fetch() дублирующий функцию postData удаляется. Т.к. используется JSON server, можно обращаться не к 'server.php' а к <http://localhost:3000/requests>.

postData('http://localhost:3000/requests', JSON.stringify(object))

Так же необходимо убрать строку .then(data => data.text()), т.к. она происходит на этапе postData и она «спрятана внутри???» (иначе срабатывает .catch() проверено). Итоговый код:

postData('http://localhost:3000/requests', JSON.stringify(object))

                .then(data => {

                    console.log(data);

                    showThanksModal(message.success);

                    statusMessage.remove();

                })

                .catch(() => {

                    showThanksModal(message.failure);

                })

                .finally(() => {

                    form.reset();

                });

Более «элегантный» способ преобразования formData:

Первоначальный код:

const object = {};

            formData.forEach(function(value, key) {

                object[key] = value;

            });

Будет использован метод formData.entries() – возвращает массив собственных перечисляемых свойств указанного объекта и метод глобального объекта Object.fromEntries(), который преобразует описанную выше структуру в обычный объект.

Пример:

const obj = {a: 23,b: 50};

console.log(Object.entries(obj));

Вернёт в консоль: [ [ 'a', 23 ], [ 'b', 50 ] ] – получаем массив с массивами (матрицу) внутри которых по два элемента, соответствующих парам ключь: значение первоначального объекта

Итоговый код:

const json = JSON.stringify(Object.fromEntries((formData.entries())));

formData превращается в массив массивов, после превращается в объект JS, после превращается в JSON. Далее этот JSON в переменной json отправляется на сервер:

postData('http://localhost:3000/requests', json)

Строим карточки с меню на основании данных с сервера:

Для получения данных для карточек будут отправляться GET запросы, отправкой GET запросов будет заниматься функция getResource – переделанная под GET функция postData. Параметр data отсутствует, т.к. ничего не отправляется на сервер, объекта с настройкой запроса так же не будет. Функция getResource просто делает запрос, дожидается его окончания и трансформирует полученные данные в объект JS. Функциональное выражение размещается после объявления класса MenuCard перед созданием экземпляров/карточек new MenuCard.

const getResource = async(url) => {

        let result = await fetch(url);

        if (!result.ok) {

            throw new Error(`could not fetch ${url}, status: ${result.status}`);

        }

        return await result.json();

    };

Решение проблемы срабатывания .cath(). Будут использованы свойства, которые возвращаются из .fetch(): 1) .ok – что-то получено и всё ок или не ок; 2) .status – статус возвращаемый сервером (200, 404, 500…). Прописывается условие – если результат не ок, код «выкинет» ошибку, если ошибка выкинута в ручном режиме, сработает блок .cath(). Для создания ошибки используется объект ошибки Error, внутрь круглых скобок помещается текст ошибки, который необходимо «выбить» в консоль. Чтобы выкинуть ошибку используется оператор throw.

Дальнейшая логика работы получаем массив с объектами, используем свойств объектов для построения карточек. Так же избавимся от повторяющегося кода создания карточек поочерёдно.

Используя функцию getResource отправляем запрос на url <http://localhost:3000/menu>, где находятся данные для формирования карточек. Возвращаемые данные обрабатываются методом .then(), зная что из getResource вернётся объект JS и даже не объект, а массив с объектами, а у массивов есть методы, которые могут перебрать элементы внутри и выполнить манипуляции с ними. Поэтому для возвращённых от сервера данных data вызывается метод .forEach() внутри которого создаётся новый экземпляр класса MenuCard для которого вызывается метод render(). Т.е. эти экземпляры будут создаваться столько раз, сколько объектов внутри массива menu (3), что приходит с сервера из файла db.json. Теперь нужно передать все данные для заполнения карточки внутрь конструктора они будут получены как свойства объекта obj из метода .forEach(), их можно записать как new MenuCard(obj.img, obj.altimg, obj.title, obj.descr, obj.price), но удобнее использовать синтаксис реструктуризации объекта – когда из объекта получаются отдельные свойства в качестве отдельных переменных, выглядит как замена obj в методе .forEach() на {img, altimg, title, descr, price}, далее эти переменные можно использовать в конструкторе.

Практика1. Создаём консольное приложение.

const numberOfFilms = +prompt('Сколько фильмов вы уже посмотрели?', '');

const personalMovieDB = {

    count: numberOfFilms,

    movies: {},

    actors: {},

    genres: [],

    privat: false

};

const key1 = prompt('Один из последних просмотренных фильмов ?', ''),

    value1 = prompt('На сколько оцените его ?', ''),

    key2 = prompt('Один из последних просмотренных фильмов ?', ''),

    value2 = prompt('На сколько оцените его ?', '');

personalMovieDB.movies[key1] = value1;

personalMovieDB.movies[key2] = value2;

console.log(personalMovieDB);

+prompt – если поставить унарный плюс перед prompt то вводимые пользователем данные будут считаться числом а не строкой.

После сообщения(message) через запятую в кавычках указывается начальное значение для поля ввода prompt, которое будет по умолчанию в нём отображаться(default), но это не обязательно.

Для записи в объект находящийся в объекте используется следующий синтаксис. Переменная в которой находится ключ передаётся объекту в квадратных скобках, после этому ключу присваивается значение.

Если не знаешь будет ли переменная изменяться по ходу программы, используй const.

Практика2

01: 'use strict';

02: const numberOfFilms = +prompt('Сколько фильмов вы уже посмотрели ?', '');

03: const personalMovieDB = {

04:     count: numberOfFilms,

05:     movies: {},

06:     actors: {},

07:     genres: [],

08:     privat: false

09: };

10: for (let i = 0; i < 2; i++) {

11:     const a = prompt('Один из последних просмотренных фильмов ?', ''),

12:         b = prompt('На сколько оцените его ?', '');

13:

14:     if (a !== null && b !== null && a !== '' && b !== '' && a.length < 50 && b.length < 50) {

15:         personalMovieDB.movies[a] = b;

16:         console.log('Ok!');

17:     } else {

18:         console.log('error');

19:         i--;

20:     }

21:

22: }

23: if (personalMovieDB.count < 10) {

24:     console.log('Просмотрено довольно мало фильмов');

25: } else if (personalMovieDB.count >= 10 && personalMovieDB.count < 30) {

26:     console.log('Вы классический зритель');

27: } else if (personalMovieDB.count >= 30) {

28:     console.log('Вы киноман');

29: } else {

30:     console.log('Произошла ошибка');

31: }

32: console.log(personalMovieDB);

В строке 14 условие задано от противного, т.е. если пользователь не нажал «отмена» на prompt(если нажата отмена он возвращает null),не оставил prompt пустым(проверка на пустую строку) и количество вводимых символов меньше 50, то всё ок и в объекте movie объекта personalMovieDB создаётся свойство(пара ключ: значение) из полученных выше данных – строка 15.

Иначе в строке 19 к итератору i применяется декремент и цикл возвращается на одну итерацию/повторение назад (снова будут заданы вопросы из prompt строка 11 и 12) и так до тех пор, пока условие не будут введены данные удовлетворяющие условия в строке 14.

Практика3

Если необходимо проверить что пользователь вводит в инпут на этапе ввода, необходимо использовать регулярные выражения.

isNaN(numberOfFilms)

Метод принимает число и проверяет, если внутри не число, возвращает true, если число false.

Паттерн – шаблон поведения.

Передача данных по ссылке или по значению, Spread оператор

Передача по значению (работает с примитивами):

let a = 5,

    b = 5;

b = b + 4;

console.log(a); //5

console.log(b); //9

Передача по ссылке (работает с более сложными структурами):

1: const obj = {

2:     a: 5,

3:     b: 1

4: };

5: const copy = obj;

6: copy.a = 10;

7: console.log(copy);

8: console.log(obj);

{ a: 10, b: 1 }

{ a: 10, b: 1 }

По логике изначальный объект obj не должен изменяться, по факту он меняется вместе с copy. Это происходит потому что в строке 5 происходит передача ПО ССЫЛКЕ, а не ПО ЗНАЧЕНИЮ, т.е. происходит передача не структуры объекта, а ссылки на изначальный объект и он же модифицируется.

1. Копию можно создать через функцию и цикл for in:

function copy(mainObj) {

    let objCopy = {};

    let key;

    for (key in mainObj) {

        objCopy[key] = mainObj[key];

    }

    return objCopy;

}

const numbers = {

    a: 2,

    b: 5,

    c: {

        x: 7,

        y: 4

    }

};

const numbersCopy = copy(numbers);

numbersCopy.a = 10;

console.log(numbersCopy); { a: 10, b: 5, c: { x: 7, y: 4 } }

console.log(numbers); { a: 2, b: 5, c: { x: 7, y: 4 }

Это называется ПОВЕРХНОСТНАЯ копия объекта – она копирует независимые свойства лежащие на первом уровне, но вложенные, как в примере объект c будут так же передаваться по ссылке.

ГЛУБОКАЯ копия – это такая копия объекта в которой независимыми являются все вложенные структуры. При её создании дублируются все свойства на пути копирования, оригинал и скопированный объект не будут иметь ничего общего.

1. Object.assign(numbers, add)

Метод Object.assign соединяет две копии объектов в одну независимую поверхностную копию объекта.

const add = {

    d: 17,

    e: 20

};

const clone = Object.assign({}, add);

clone.d = 23;

console.log(add); // { d: 17, e: 20 }

console.log(clone); // { d: 23, e: 20 }

С помощью этого метода можно создавать поверхностную копию объекта, передав методу в аргументе пустой объект.

1. Создание копии массива:

const oldArray = ['a', 'b', 'c'];

const newArray = oldArray.slice();

newArray[1] = 'q';

console.log(oldArray);  //[ 'a', 'b', 'c' ]

console.log(newArray);  //[ 'a', 'q', 'c' ]

.slice – метод позволяющий скопировать старый массив, в него можно передать аргументы по количеству элементов.

1. Создание поверхностной копии с помощью оператора разворота Spread:

В ES6 этот оператор появился для массивов, а в ES8 (стандартизирован в ES9 2018 г.) для объектов.

function log(a, b, c) {

    console.log(a);

    console.log(b);

    console.log(c);

}

const num = [2, 5, 7];

log(...num);

В данном случае оператор spread(…) служит для раскладывания массива num на 3 отдельных элемента, которые могут быть переданы в аргумент функции log.

const array = ['a', 'b', 'c'];

const arr = [...array];

arr[0] = 'www';

console.log(array);     //[ 'a', 'b', 'c' ]

console.log(arr);       //[ 'www', 'b', 'c' ]

Новому массиву arr присваивается результат раскрытия массива array, в результате получается независимая поверхностная копия.

const obj1 = {

    one: 1,

    two: 2,

    three: 3

};

const obj2 = {...obj1};

obj2.one = 4;

console.log(obj1);      //{ one: 1, two: 2, three: 3 }

console.log(obj2);      //{ one: 4, two: 2, three: 3 }

Тот же принцип для объектов.

Практика 4

Рефакторинг кода – переписывание его под новые условия и новые задачи.

console.log() – сама по себе ничего не возвращает, в ней нет ключевого слова return, поэтому она выводит undefined когда запускается внутри браузера. Такое же поведение может наблюдаться у функций и методов запускаемых в браузере и не имеющих команду return.

Метод writeYourGenres можно реализовать двумя способами:

 writeYourGenres: function() {

        for (let i = 1; i <= 3; i++) {

           let genre = prompt(`Ваш любимый жанр под номером ${i}`).toLowerCase();

            if (genre == '' || genre == null) {

                console.log('Введены некорректные данные или ничего не введено');

                i--;

            } else {

*this*.genres[i - 1] = genre;

            }

        }

*this*.genres.forEach((item, i) => {

            console.log(`Любимый жанр #${i+1} - это ${item}`);

        });

    }

1) Вначале задаётся цикл for, который будет повторяться 3 раза. В переменную genre помещается строка полученная из функции prompt преобразованная в нижний регистр.

Если строка в genre пустая или пользователь нажал отмена, в консоль выводится сообщение об ошибке и цикл возвращается на одну итерацию назад. Иначе, если предыдущие условия не выполнились, пустому массиву, находящемуся в свойстве genres, под индексом i-1 присваивается полученная выше строка.

Методом .forEach осуществляется перебор массива, находящегося в свойстве genres объекта personalMovieDB. В результате чего в консоль выводится строка указанного содержания.

writeYourGenres: function() {

        for (let i = 1; i < 2; i++) {

let genres = prompt(`Введите ваши любимые жанры через запятую`).toLowerCase();

            if (genres == '' || genres == null) {

                console.log('Введены некорректные данные или ничего не введено');

                i--;

            } else {

*this*.genres = genres.split(', ');

*this*.genres.sort();

            }

        }

*this*.genres.forEach((item, i) => {

            console.log(`Любимый жанр #${i+1} - это ${item}`);

        });

2) Вначале задаётся цикл for, который будет повторяться 1 раз(т.к. мы запрещаем пользователю нажимать кнопку отмена или оставлять пустую строку, а корректные результаты ввода будут сразу помещены в свойство .genres).

Создаётся переменная genres в которую помещается результат вызова функции prompt (пользователю задаётся вопрос с просьбой ввести необходимую информацию, все введённый данные будут преобразованы в нижний регистр методом .toLowerCase()).

Проверяется условие, если в переменной genres пустая строка или null(нажата отмена), в консоль выводится сообщение об ошибке и цикл возвращается на одну итерацию назад(пользователю вновь задаётся вопрос, вновь проверяется условие до тех пор пока введённые данные не будут корректны), иначе свойству .genres массива personalMovieDB присваивается массив, получаемый в результате резделения методом .split() строки, содержащейся в переменной genres, разделение происходит после ‘, ’.

С помощью метода .sort() массив находящийся в свойстве genres сортируется в алфавитном порядке.

С использованием метода .forEach() массив перебирается, callback функция выводит в консоль сообщение, где для удобства восприятия пользователем {i+1} – это индекс элемента плюс единица, НО перебор всё равно начнётся с элемента с индексом 0. ВАЖНО i – это индекс элемента а не итерация цикла, перебор осуществляется после того как предыдущий цикл for уже отработал и его итератор уже перестал существовать.

Отлавливаем ошибки при помощи консоли разработчика. Breakpoints

Breakpoint – это метка во вкладке source консоли разработчика, которая позволяет останавливать код в определённом участке. Устанавливается методом клика по номеру строки, после чего нужно перезагрузить страницу.

Дебаг можно включать прямо из кода с помощью ключевого слова debubber(“нативный breakpoint”).

function hello() {

    console.log("hello world!");

    debugger;

}

Задачи с собеседования на понимание основ

1. Какое будет выдано значение?

let x = 5;

alert( x++ );

Выдаст 5, потому что постфиксная форма х++ возвращает значение переменной увеличенное на единицу, но перед тем как это сделать, она возвращает исходное значение. Т.е. сначала будет выдана 5, а уже ПОСЛЕ значение переменной увеличится.

alert(++x);

Вернёт 6, потому что префиксная форма инкремента возвращает уже ИЗМЕНЁННОЕ значение.

Будет выдано значение 5.

1. Чему равно такое выражение?

[ ] + false - null + true;

Вначале его следует разбить на составные части:

console.log(typeof([] + false));

Выдаст string. ОСОБЕННОСТЬ - когда происходит подобное взаимодействие с пустым массивом, он будет преобразован в строку. Если сложить строку и какое либо значение, получается строка, значит результатом этой части выражения будет “false”.

console.log([] + false-null);

Выдаст NaN, т.к. происходит выполнение нематематических операций, а именно сложение строчного типа данных с числовым.

console.log([] + false - null + true);

Выдаст NaN(конечный результат), т.к. от прибавления булинового значения результат выражения не меняется.

Выражение равно NaN.

1. Что выведет этот код ?

let y = 1;

let x = y = 2;

alert(x);

Происходит последовательное присваивание. Т.к. число – это примитивный тип данных, то оно передаётся по значению. Передача идёт с права на лево: сначала переменной у присваивается значение 2, после этого значение переменной у(2) присваивается переменной х.

Код выдаст 2.

1. Чему равна сумма ?

[] + 1 + 2;

Поэтапно:

- вначале пустой массив преобразуется в строку,

- затем происходит конкатенация строки и числа, результатом будет строка «1»,

-после вновь происходит конкатенация строки «1» с числом 2.

Результатом выполнения кода будет строка «12».

1. Что выдаёт код ?

alert("1"[0]);

Решение заключается в принципе, что к каждому элементу/символу строки можно обратиться по его индексу! Если строка состоит лишь из одного символа «1», то символом этой строки с индексом [0] будет 1.

Алерт выдаст строку 1.

1. Чему равно выражение ?

2 && 1 && null && 0 && undefined;

У оператора И && есть особенность относительно того, какое выражение он может вернуть – оператор И && ВСЕГДА запинается на лжи.

ВАЖНО: аналогичное поведение и у оператора ИЛИ || только он всегда запинается на правде!

Разъяснение пошагово: проверка происходит с лева направо – 2, в логическом контексте это псевдоистина, 1, аналогично, null – псевдолож и т.к. оператор И && впервые запнулся на null, то его он и будет возвращать. Дальнейшая проверка проводиться не будет, т.к это больше не нужно. Результат выполнения выражения можно сравнить с работой оператора return с первым встретившимся ложным выражением.

Выражение равно null.

1. Есть ли разница между выражениями ?

!!( a && b ) и (a && b);

Выражение приводится в более понятный вид, методом подставления числовых значений:

console.log(!!( 1 && 2 ) === (1 && 2));

Два восклицательных знака !! превращают идущее за ними выражение в булиновое:

console.log(!!( 1 && 2 ));

даст true

Вторая часть выражения

console.log(( 1 && 2 ));

даст число 2 т.к. если выражение истинно, оператор И && возвращает последнее встретившееся истинное/псевдоистинное выражение. 2 – в этом случае это псевдоистина, как и 1.

Сравнение полученных выражений

console.log(true === 2);

даст false.

Следовательно между выражениями есть разница.

Ремарка: если бы сравнение было не строгим ===, а с преобразованием типов == и операторы в последних скобках были поменяны местами, то выражение было бы истинно:

console.log(!!( 1 && 2 ) == (2 && 1));

даёт true.

1. Что выдаёт этот код ?

alert( null || 2 && 3 || 4 );

Когда нет уверенности какой из операторов сработает первым, необходимо смотреть таблицу приоритетов операторов. Из таблицы следует, что Логическое И имеет приоритет 6, а Логическое ИЛИ 5, следовательно И выполнится раньше, т.е. приоритет выше.

2 && 3

результатом работы И будет число 3, т.к. 2 и 3 в логическом смысле псевдоистина, значит выражение правдиво и будет возвращено последнее значение.

Далее сравнение идёт слева на право:

null || 2 && 3 // null || 3

вернёт 3 т.к. оператор ИЛИ || запинается на правде и вернёт первую встреченную правду – результат проверки выражения 2 && 3 (3).

Далее проверяется правая часть:

3 || 4

т.к. и 3 и 4 псевдоистина, но ИЛИ || запинается на первой встреченной правде, вернётся 3.

Код выдаёт 3.

1. Правда ли что a == b ?

const a = [1, 2, 3],

    b = [1, 2, 3];

Результатом будет false, т.к. a и b – это два разных хранилища информации (не одно и то же!), которые в данный момент содержат одинаковые данные.

Массив a не равен массиву b.

1. Что выведет этот код ?

alert( +"Infinity" );

Т.к. перед строкой “Infinity”стоит унарный плюс то выводимое значение будет числового типа.

Код выведет Infinity с типом данных число.

1. Верно ли сравнение?

"Ёжик" > "яблоко";

Сравнивать две строки – это нормально, в таком случае идёт посимвольное сравнение, регистр буков так же учитывается, т.к. буквы разного регистра – это разные буквы. При подобных сравнениях нужно учитывать особенности юникода(существует таблица юникода, где все символы идут под определённом номером).

Таким образом сравнение выдаст false. Сравнение неверно.

1. Чему равно выражение?

0 || "" || 2 || undefined || true || falsе;

В логическом контексте: 0-false, “”(пустая строка)-false, 2-true, undefined-false, true-true, false-false. Оператор ИЛИ || запинается на первой встреченной правде и возвращает её, следовательно он запнётся на 2 и вернёт её.

Выражение равно 2.

Практика 5 отработка действий со странице

Все пути и путь к картинке в частности прописывается относительно index.html, т.к. JS код исполняется на html страничке.

Для того, чтобы полностью очистить элемент на странице, можно воспользоваться свойством .innerHTML и передать в него пустую строку:

movieList.innerHTML = "";

+= это дополнительное присваивание, подразумевается что a = a+1 идентично a+=1. Аналогично работает и со строками a = a + “abc” идентично a += “abc”.

Свойство .innerHTML так же позволяет получать элементы со страницы следующим способом:

console.log(poster.innerHTML);

выдаст в консоль вёрстку, которая находится в poster. Используется редко.

/\* Задания на урок:

1) Удалить все рекламные блоки со страницы (правая часть сайта)

2) Изменить жанр фильма, поменять "комедия" на "драма"

3) Изменить задний фон постера с фильмом на изображение "bg.jpg". Оно лежит в папке img.

Реализовать только при помощи JS

4) Список фильмов на странице сформировать на основании данных из этого JS файла.

Отсортировать их по алфавиту

5) Добавить нумерацию выведенных фильмов \*/

10: 'use strict';

11:

12: const movieDB = {

13:     movies: [

14:         "Логан",

15:         "Лига справедливости",

16:         "Ла-ла лэнд",

17:         "Одержимость",

18:         "Скотт Пилигрим против..."

19:     ]

20: };

21:

22: const adv = document.querySelectorAll('.promo\_\_adv img'), //1

23:     poster = document.querySelector('.promo\_\_bg'), //3

24:     genre = poster.querySelector('.promo\_\_genre'), //2

25:     movieList = document.querySelector('.promo\_\_interactive-list'); //4-5

26:

27: adv.forEach(item => {   //1

28:     item.remove();

29: });

30:

31: genre.textContent = 'драма';    //2

32:

33: poster.style.backgroundImage = "url('img/bg.jpg')";     //3

34:

35: movieList.innerHTML = "";   //4

36:

37: movieDB.movies.sort();      //4

38:

39: movieDB.movies.forEach((item, i) => {           //4-5

40:     movieList.innerHTML += `

41:     <li class="promo\_\_interactive-item">${i + 1} ${item}

42:         <div class="delete"></div>

43:     </li>

44:     `;

45: });

В строке 22 используется возможность получения элемента с помощью .querySelectorAll по комбинированному селектору. Используется класс и тэг элемента.

Выполнение заданий 4-5: в строке 25 получаем родителя <ul> со списком элементов <li> внутри. В строке 35 очищаем его внутренний HTML. В строке 37 происходит сортировка массива movieDB в алфавитном порядке согласно требованиям задания 4. В строке 39 для свойства(массива) объекта movieDB.movies(!!! не для movieList !!!, для него метод не сработает из-за способа получения и дальнейшего смысла в этом нет) применяется метод перебора forEach с параметрами item – фильм, i – итератор цикла. Строка 40: используя свойство .innerHTML на каждой итерации цикла будет происходить дополнительное присваивание элементу <ul> который находится в переменной movieList, html разметки, которая находится в строках 41-43. Благодаря += внутренний html не будет перезаписываться, а будет дополняться новыми строками с каждой итерацией цикла.

Практика 6 Используем события на странице проекта

/\* Задания на урок:

1) Реализовать функционал, что после заполнения формы и нажатия кнопки "Подтвердить" -

новый фильм добавляется в список. Страница не должна перезагружаться.

Новый фильм должен добавляться в movieDB.movies.

Для получения доступа к значению input - обращаемся к нему как input.value;

P.S. Здесь есть несколько вариантов решения задачи, принимается любой, но рабочий.

2) Если название фильма больше, чем 21 символ - обрезать его и добавить три точки

3) При клике на мусорную корзину - элемент будет удаляться из списка (сложно)

4) Если в форме стоит галочка "Сделать любимым" - в консоль вывести сообщение:

"Добавляем любимый фильм"

5) Фильмы должны быть отсортированы по алфавиту \*/

В JS существует событие load – срабатывает когда страница максимально готова к работе. Но если на странице есть тяжелые скрипты или картинки, событие load может сработать через 5-10 секунд и пользователю придётся прождать это время. Поэтому это событие используется не всегда.

Чаще всего используется событие DOMContentLoaded – срабатывает когда построена DOM структура, не нужно ждать полной загрузки страницы. Сформированного DOM достаточно для выполнения основных скриптов, а картинки/другие скрипты/шрифты/стили могут подгружаться фоново. Синтаксис следующий:

'use strict';

document.addEventListener('DOMContenLoaded', () => {

    //Сюда помещаем весь код

});

Если необходимо использовать событие DOMContentLoade, это делается через обработчик событий, а в callback функцию помещается ВЕСЬ! код, который до этого был на странице. Вместо document может быть window но разницы в работе не будет.

addForm.addEventListener('submit', (event) => {

        event.preventDefault();

        const newFilm = addInput.value;

        const favorite = checkbox.checked;

    });

Обработчик навешивается на форму и срабатывает по событию отправки формы submit. В callback функции происходит отмена стандартного поведения браузера(т.е. перезагрузка страницы при нажатии кнопки отправки формы). Создаётся переменная newFilm в которую с помощью свойства .value помещается вводимое в инпут значение. Создаётся переменная favorite в которую с помощью свойства .checked помещается состояние чекбокса (true / false) checkbox заранее полученного со страницы.

«Правильное» программирование – когда нет жестко заданных элементов, и функция узнаёт с чем она будет работать только в момент вызова, т.е. может быть использована многократно но с разными аргументами.

function createMovieList(films, parent) {

        parent.innerHTML = "";

        films.forEach((item, i) => {

            parent.innerHTML += `

            <li class="promo\_\_interactive-item">${i + 1} ${item}

                <div class="delete"></div>

            </li>

            `;

        });

    }

createMovieList(movieDB.movies, movieList);

Все действия по созданию элементов на странице(практика 5), кроме сортировки оборачиваются в функцию createMovieList, с условными параметрами films – фильмы (заменит movieDB.movies - массив с фильмами внутри объекта) и parent (заменит movieList – список фильмов на странице) родительский элемент куда будет вставлен html код с фильмами – элементами списка. После объявления функции она вызывается для формирования первичного списка из фильмов изначально прописанных в movieDB.movies, вторым аргументом будет movieList – элемент <ul> в который будут помещены динамически формируемые <li> со введёнными пользователем фильмами.

const deleteAdv = (arr) => {

        arr.forEach(item => {

            item.remove();

        });

    };

    deleteAdv(adv);

Удаление рекламных блоков переписывается в виде функционального выражения с использованием синтаксиса стрелочной функции. Таким образом убирается жёсткая привязка к конкретному псевдомассиву элементов и функция может быть повторно использована с другими условными массивами элементов, которые необходимо удалить со страницы. Для начала работы функция должна быть вызвана с заранее полученным псевдомассивом в качестве аргумента.

const makeChanges = () => {

        genre.textContent = 'драма';

        poster.style.backgroundImage = "url('img/bg.jpg')";

    };

    makeChanges();

Мелкие изменения на странице так же оборачиваются в функцию без параметров, т.к. пока это не целесообразно. Чтобы изменения применились, функция вызывается.

const sortArr = (arr) => {

        arr.sort();

    };

    sortArr(movieDB.movies);

Сортировка массива с фильмами внутри объекта movieDB будет производиться заранее созданной функцией sortArr, выполняющей сортировку переданного ей массива по алфавиту.

Принято сначала объявлять ряд функций, а уже в конце кода их вызывать. Не стоит забывать, что если функция объявлена с помощью функционального выражения, то она будет обработана со всем кодам и вызвать её раньше чем она объявлена не получится.

31:     addForm.addEventListener('submit', (event) => {

32:         event.preventDefault();

33:         const newFilm = addInput.value;

34:         const favorite = checkbox.checked;

35:

36:         movieDB.movies.push(newFilm);

37:         sortArr(movieDB.movies);

38:

39:         createMovieList(movieDB.movies, movieList);

40:         event.target.reset();

41:     });

Для того, чтобы на странице добавлялись новые фильмы по мере их ввода в форму, внутри callback функции обработчика событий формы в строке 39 вызывается функция createMovieList, которая будет генерировать их на странице. После ввода необходимо очистить форму это реализуется с помощью метода reset применённого к элементу объекта события event.target.

Во избежание заполнения списка пустыми строками(когда submit жмётся при незаполненном инпуте) нужно произвести проверку на наличие значения:

addForm.addEventListener('submit', (event) => {

        event.preventDefault();

        const newFilm = addInput.value;

        const favorite = checkbox.checked;

        if (newFilm) {

            movieDB.movies.push(newFilm);

            sortArr(movieDB.movies);

            createMovieList(movieDB.movies, movieList);

        }

        event.target.reset();

    });

Если в newFilm НЕ пустая строка = true, то введённый фильм добавляется в массив movies, функция sortArr сортирует массив movies, а функция createMovieList динамически формирует список фильмов. Если инпут оставлен пустым, пустая строка = false, то ничего не происходит. Первое задание выполено!

addForm.addEventListener('submit', (event) => {

        event.preventDefault();

        let newFilm = addInput.value;

        const favorite = checkbox.checked;

        if (newFilm) {

            if (newFilm.length > 21) {

                newFilm = `${newFilm.substring(0, 22)}...`;

            } // дальше код тот же что и выше

Для выполнения второй задачи используется проверка условия – если длина строки в newFilm больше 21 символа, переменной newFilm присваивается значение подстроки полученной после применения к newFilm метода .substring от 0 до 22 символа не включая его, строка завершается троеточием. Для присваивание newFilm нового значения тип этой переменной предварительно меняется на let. Второе задание выполнено!

Т.к. изображение корзинки – это <div> с классом .delete внутри <li>, то для удаления элемента при клике на корзину, нужно удалять её родителя, т.е. <li>, а так же удалять фильм из массива movieDB.movies.

65:     function createMovieList(films, parent) {

    66:         parent.innerHTML = "";

    67:         sortArr(films);

    68:

    69:         films.forEach((item, i) => {

    70:             parent.innerHTML += `

    71:             <li class="promo\_\_interactive-item">${i + 1} ${item}

    72:                 <div class="delete"></div>

    73:             </li>

    74:             `;

    75:         });

    76:

    77:         document.querySelectorAll('.delete').forEach((btn, i) => {

    78:             btn.addEventListener('click', () => {

    79:                 btn.parentElement.remove();

    80:                 movieDB.movies.splice(i, 1);

    81:

    82:                 createMovieList(films, parent);

    83:             });

    84:         });

    85:     }

Корзины больше нигде использоваться не будут, поэтому получаем их в строке 77 прямо внутри функции createMovieList, тут же перебираем их с помощью метода .forEach для того чтобы навесить один обработчик на несколько элементов. Callback функция передаваемая .forEach() имеет 2 параметра, btn – кнопка-корзина, i - нумерация элементов массива с фильмами.Строка 78 - внутри функции каждой корзинке навешивается обработчик события click и callback функция, внутри которой в строке 79 происходит удаление родительского элемента корзины со страницы, а в строке 80 к массиву movieDB.movies применяется метод .splice(i, 1) (i – индекс только-что удалённого элемента с которого следует начать, 1 – количество элементов, которые нужно удалить) для удаления фильма из массива/базы данных.

Для того чтобы нумерация фильмов так же менялась при удалении фильмов, используется приём рекурсии, т.е. функция вызывает сама себя. Когда происходит клик по корзине = удаление элемента, необходимо, чтобы элементы списка формировались заново. За это отвечает функция createMovieList, которая вызывается в строке 82 внутри себя же с аргументами films = movieDB.movies и parent = movieList. Третье задание выполнено!

Выполнения задания № 5 – при удалении, сортировка работает некорректно. Для избегания этого в строке 67 производится вызов функции sortArr внутри функции createMovieList, чтобы элементы списка сортировались в процессе создания. Вызов sortArr вне этой функции удаляется, т.к. при загрузке страницы список фильмов ненужно заново сортировать. Теперь createMovieList будет каждый раз вызывать sortArr и даже при реализации «удаления корзинкой» список будет повторно сортироваться. Пятое задание выполнено!

31:     addForm.addEventListener('submit', (event) => {

    32:         event.preventDefault();

    33:         let newFilm = addInput.value.toUpperCase();

    34:         const favorite = checkbox.checked;

    35:

    36:         if (newFilm) {

    37:

    38:             if (newFilm.length > 21) {

    39:                 newFilm = `${newFilm.substring(0, 22)}...`;

    40:             }

    41:

    42:             if (favorite) {

    43:                 console.log("Добавляем любимый фильм");

    44:             } //Дальше код прежний

Для выполнения задания № 4 проверяется содержимое переменной favorite, объявленной в строке 34. Если условие в строке 42 выполняется (favorite = true), то при отмеченном чекбоксе выводится соответствующее сообщение в консоль, если чекбокс не отмечен=условие не выполнено=false, ничего не происходит. Для того, чтобы на сортировку не влиял регистр букв, добавил в строку 33 вызов к значению получаемому из инпута метода .toUpperCase(). Четвёртое задание выполнено!