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)

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.

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

15. 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.

При работе с массивом .length возвращает не количество существующих элементов в массиве, а индекс последнего элемента+1.

.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

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

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

.split(‘,’) – разбивает строку на массив путём разделения после указанного символа (подсроки ‘,’).

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

.sort() – метод сортирует массив по алфавиту, используется для строковых значений. 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.

У псевдомассива нет вышеперечисленных методов.

ДИНАМИЧЕСКАЯ ТИПИЗАЦИЯ

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

Превращение в число:

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

Преобразование в Булево значение:

1. 0, ‘’, null, undefined, NaN – псевдолож, всё остальное псевдоистина.
2. Boolean(…) используется редко
3. !! перед любым выражением используется редко

ПОЛУЧЕНИЕ ЭЛЕМЕНТОВ СО СТРАНИЦЫ

document.getElementsByTagName(‘div’) – получает псевдомассив (коллекцию) с элементами, после с помощью индекса можно получить определённый элемент; css селекторы задаются в скобках.

document.getElementsByClassName(‘circle’) – аналогично.

document.querySelectorAll(‘.heart’) – самый популярный метод, поддерживает вложенность. Псеводмассив полученный с помощью этого метода имеет один метод – for each, позволяющий перебрать его и выполнить действия с каждым из его элементов. css селекторы задаются в скобках с указанием типа(как в css, прим. . – класс, # - id и т.д.).

document.querySelector(‘.heart’) – получает первый элемент на странице по выбранному селектору.

ДЕЙСТВИЯ С ЭЛЕМЕНТАМИ НА СТРАНИЦЕ

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.

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).

document.createElement(‘div’) – создаёт элемент.

document.createTextNode(‘dffddffd’) – создаёт текстовый узел без html тэга.

Для стилизации созданных в JS элементов страницы обычно используются заранее созданные в css классы, а не множество строк .style. …

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.

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. .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.

document.body.replaceChild(btn[1], circle[1]);

заменяет один заранее полученный элемент другим заранее полученным элементом.

div.innerHTML = "<h1>Hello!</h1>";

создание html разметки 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.

СОБЫТИЯ И ИХ ОБРАБОТЧИКИ

1. Вписать код прямо в html разметку(почти не используется, только для маленького кода):

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

1. Получить 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. Функции продолжают работать бесконечно(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.

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, чтобы например, скрыть элемент:

let target = event.target;

target.style.display = 'none';

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

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

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

    event.preventDefault();

});

Отменяет стандартное поведение браузера относительно объекта event, например отменяет переход по ссылке.

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 с псевдомассив кнопок с помощью .querrySelectorAll, назначаем каждому item(кнопке) обработчик

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

СОБЫТИЯ НА МОБИЛЬНЫХ УСТРОЙСТВАХ

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.

event.touches

объект содержащий все прикосновения к сенсору.

event.changedTouches

аналогично даёт список касаний провзаимодействовавших с устройством.

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.

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 \.

УПРАВЛЕНИЕ ВРЕМЕНЕМ ПРИМИНЕНИЯ СКРИПТОВ

let timer = setTimeout(sayHello, 3000);

clearTimeout(timer);

setTimeout – устанавливает таймер выполнения события (функции sayHello) через 3000 милисекунд. clearTimeout – отменяет выполнение таймера в обработчике timer.

let timer = setInterval(sayHello, 3000);

clearInterval(timer);

То же, что и выше, но событие повторяется через указанный интервал.

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 функции произойдёт сразу же. Для решения проблемы используется рекурсивный вызов setTimeout (функция вызывает сама себя).

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.

ПАРАМЕТРЫ ДОКУМЕНТА, ОКНА И РАБОТА С НИМИ

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. Для ширины это не страшно, т.к. скролы с горизонтальной прокруткой почти не используются.

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

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

});

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

Изменяемые свойства:

.scrollTop .scrollLeft – содержат ширину и высоту невидимой, уже прокрученной части элемента сверху или слева.

13. Callback функции

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

14. Объекты

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

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

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

Object.keys(имя\_объекта).length – количество свойств объекта.

15. Массивы и псевдомассивы

16. ООП

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

Даты в JS

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

<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 милисекунд.

24\*60\*60\*1000

Количество милисекунд в сутках: 24 часа \* 60минут в часе \* 60 секунд в минуте \* 1000 милисекунд 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 милисекунды 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 милисекунд в дату формата ISO, .substring оставляет от неё год-месяц-день, <br> - каждая дата с новой строки.

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.

Анимация в JS

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

Делегирование событий

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. Если цель клика это кнопка и имя тэга элемента BOTTON, в консоль выводится соответствующее сообщение. Клики по элементу родителю ничего не выводят. Т.е. если элемент подходит под условия, то на него будет работать написанная или переданная функция, действия с родителя делегируются на его потомков. Можно добавлять любое количество потомков, код будет работать.

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

проверяет есть у элементов-целей события класс ‘first’

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

ищет среди элементов-целей совпадения – элемент button с классом .first

Параметры документа, окна и работа с ними

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.

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 возвращающий объект, содержащий bottom, top, left, right, width, x, y.

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:

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

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

Конструкторы и классы

Конструкторы использовались в стандарте ES5, в ES6 используются классы.

Синтаксис 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');

};

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, 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 объект Window (глобальный объект в вэб браузере)

'use strict'

function showThis(a, b) {

    console.log(*this*);

    function sum() {

        console.log(*this*);

        return a + b;

    }

    console.log(sum());

}

showThis(4, 5);

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

let obj = {

    a: 20,

    b: 15,

    sum: function() {

        console.log(*this*);

    }

};

obj.sum();

Вызов метода sum объекта obj даёт сам объект. Контекст выполнения – 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, 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 свойств будет только что созданный объект, т.е. this ссылается на новосозданный объект.

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. Но использование метода .call(если в качестве параметра передаётся одна строка) и метода .apply(если в качестве параметра нужно передать массив из нескольких значений) присваивают this значение объекта, который был передан им в качестве аргумента.

function count(number) {

    return *this* \* number;

}

let double = count.bind(2);

console.log(double(3));

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:

et 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 обработчика, то контекстом вызова будет тот элемент на котором происходит событие (в данном случае <button>, при клике её ширина станет 200px). НО если объявить функцию внутри функции-обработчика и вызвать, при её контекстом вызова останется глобальный объект Window.

Стандарт ES6 (наиболее часто используемые)

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.

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:

function calcOrDouble(number, basis) {

    basis = basis || 2;

    console.log(number \* basis);

}

calcOrDouble(3, 5);

calcOrDouble(6);

Актуально в ES5: 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.

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.

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

function foo(...args) {

    console.log(args);

}

foo(1, 2, 3, 4, 5); // [1, 2, 3, 4, 5]

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

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

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.

AJAX

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 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.

Методы XMLHttpRequest:

request.open();

1)Метод open чаще всего идёт сразу за созданием объекта XMLHttpRequest, производит 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);

async – асинхронность запроса, по умолчанию true, если указать false то пока сервер не ответит со страницей нельзя будет взаимодействовать(зависнет);

login – логин;

pass – пароль.

request.open('GET', 'current.json');

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

HTTP запросы:

Обычно состоят из двух частей – head и body.

Когда отправляются AJAX запросы, можно задать их настройки – что это за запрос, что он содержит и т.д.

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, а это будет json в кодировке utf-8.

request.send(body);

3)Метод send открывает соединение и отправляет запрос на сервер. В качестве аргумента, он принимает body – тело HTTP запроса, НО тело у запросов бывает только тогда, когда информация берется из клиентской части и отправляется на серверную(пример – форма обратной связи на сайте).

request.send();

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

Свойства XMLHttpRequest:

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

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).

load – срабатывает когда запрос полностью загрузился без ошибок.

let inputRub = document.getElementById('rub'),

    inputUsd = document.getElementById('usd');

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

    let request = new XMLHttpRequest();

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

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

    request.send();

    request.addEventListener('readystatechange', function() {

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

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

            inputUsd.value = inputRub.value / data.usd;

        } 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: http 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 , 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. Если условие не верно, в консоль выведется сообщение об ошибке.

Promise

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

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.

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.