**Рабочее пространство**

VSCode <https://code.visualstudio.com/>

Список необходимых плагинов для VSCode:

1. All Autocomplete
2. Auto Close Tag
3. Auto Complete Tag
4. Auto Rename Tag
5. Beautify
6. Code Runner
7. Import Cost
8. JavaScript (ES6) code snippets
9. jshint
10. Live Server
11. Multiple clipboards for VSCode
12. Reactjs code snippets
13. Sass
14. Theme - Oceanic Next
15. vscode-icons
16. ESLint

VSCode -> Emmet :

|  |  |
| --- | --- |
| ! <Tab> | <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <meta http-equiv="X-UA-Compatible" content="IE=edge">      <meta name="viewport" content="width=device-width, initial-scale=1.0">      <title>Document</title>  </head>  <body> |
| .class < Tab > | <div class="class"></div> |
| .class>button < Tab > | <div class="class"><button> </button></div> |
| .class>button>div < Tab > | <div class="class">      <button>          <div></div>      </button>  </div> |

Плагин jshint

Конфиг плагина jshint – файл .jshintrc, помещается в папку проекта или в папку выше, которая включает проекты, где должен действовать конфиг.

Окружение Node.js

<https://nodejs.org/en/>

**4. Подключение jscript**

На уровне файла index.html создаём папку js/ и в ней файл script.js.

В html перед закрывающим тегом body:

<body>

    <p> Hello!</p>

…

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

</body>

Атрибут тега script

type="text\javascript"

устаревший. Его надо удалять.

Проверка:

а) На веб-странице F12 или ПКМ-«Исследовать элемент». Вкладка Console. Там вывод js.

б) Или консоль в VSCode – вкладка Output. Выделить нужную часть кода, ПКМ-Run code. В Output’е видим вывод.

**5.1 Объявление переменных**

По стандарту ES6

let number = 5;

const width = 1;

Имена регистрозависимые.

Вариант

var name = Ivan;

устаревший. Не использовать.

let и const не подвержены всплытию переменной, как var, видны только после объявления и в блоке кода, ограниченном фигурными скобками { и }.

Новый стандарт не поддерживается старыми браузерами. Проверить применимость (поддержку возможностей) конструкций языка на сайте <https://caniuse.com/>.

**5.2 Режим строгого современного кода**

Директива

"use strict";

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode>

**5.3 Стилистические правила**



**6. Классификация типов данных в JavaScript**

****

<https://medium.com/@hydrock/bigint-новый-тип-данных-в-js-dd5c29446570>

<https://learn.javascript.ru/symbol>

//number

let number = 4.6;

//string

let name = 'Alex';

let name = "Alex";

let name = `Alex`;

//null

console.log(result);

//undefined

let und;

console.log(und);

//object

const obj = {

    name: "John",

    age: 25,

    isMarried: false

};

console.log(obj.name);

console.log(obj["name"]);

const emptyObj = {}; //пустой объект

//array

let arr = ['plum.png', 'orange.jpg', 6, 'apple.bmp'];

console.log(arr[1]);  // orange.jpg

const emptyArr = []; //пустой массив

**7. Простое общение с пользователем**

Alert

alert('Привет!');



Confirm

const result = confirm('Are You here?');



const answer = prompt("Как вас зовут?", "");



Если пользователь нажмет Отмена, то answer=null.

Все ответы пользователя имеют тип данных string. Если пользователь ничего не введёт, то answer=’’.

Проверка типа данных:

console.log(typeof(answer));

string

(https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/typeof)

Для преобразования в число можно использовать оператор +

const answer = +prompt("Сколько вам лет?", "");

console.log(typeof(answer));

number

Заполнение массива ответами пользователя

const answers = [];

answers[0] = prompt("Ваше имя?", "");

answers[1] = prompt("Ваша фамилия?", "");

answers[2] = prompt("Ваш возраст?", "");

document.write(answers);



console.log(typeof(answer));

object

(<https://developer.mozilla.org/ru/docs/Web/API/Document/write> )

Команда

document.write(answers);

устаревшая, полностью заменяет содержимое страницы.

Однако alert, prompt и т.п. блокируют построение страницы, поэтому Hello! остаётся.

**8. Интерполяция (ES6)**

- вставка значения переменной в строку.

Используется вместо конкатенации строк.

Например, два алерта идентичны

const user = "Ivan";

alert('Привет, ' + user + '!');

alert(`Привет, ${user}!`);

В строке, использующей интерполяцию обязательно использование обратных кавычек (на клавише Ё) (см. второй алерт).

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals>

**9. Операторы в JS**

+, -, \*, / - арифметические

Конкатенация

console.log('4' + "5");

// expected 45

Унарный плюс (преобразует тип string к number)

console.log(4 + +'5');

// expected 9

Инкремент, декремент (постфиксная форма)

let a = 10,

    b = 10;

a++;

b--;

console.log(a);

// expected 11

console.log(b);

// expected 9

Инкремент, декремент (префиксная форма)

let a = 10,

    b = 10;

++a;

--b;

console.log(a);

// expected 11

console.log(b);

// expected 9

Различие этих форм в выполнении «на месте»:

Постфиксная форма сначала возвращает исходное значение

let a = 10,

    b = 10;

console.log(a++);

// expected 10

console.log(b--);

// expected 10

Префиксная форма

let a = 10,

    b = 10;

console.log(++a);

// expected 11

console.log(--b);

// expected 9

Остаток от деления

console.log(8%3);

// expected 2

=

Присваивание

let a = 10;

Сравнение

console.log(2\*4 == 8);

// expected true

Строгое сравнение (по значению и по типу данных)

console.log(2\*4 == "8"); //число сравнивается со строкой

// expected true

console.log(2\*4 === "8"); //число сравнивается со строкой

// expected false

Логическое И и ИЛИ

let isChecked = true,

    isClosed = false;

console.log(isChecked || isClosed); //ИЛИ

// expected true

console.log(isChecked && isClosed); //И

// expected false

Отрицание !

console.log(isChecked && !isClosed);

// expected true

Не равно !=

console.log(6 != 8);

// expected true

console.log(6 != '6');

// expected false

console.log(6 !== '6');

// expected true

!= - нестрогое сравнение,

!== - сравнение строгое по типу.

[Таблица](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Operator_Precedence#table) приоритетов операторов

Данная таблица упорядочена с самого высокого приоритета (20) до самого низкого (1).

|  |  |  |  |
| --- | --- | --- | --- |
| **Приоритет** | **Тип оператора** | **Ассоциативность** | **Конкретные операторы** |
| 20 | [Группировка](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Grouping) | не определено | ( … ) |
| 19 | [Доступ к свойствам](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Property_Accessors#dot_notation) | слева направо | … . … |
| [Доступ к свойствам с возможностью вычисления](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Property_Accessors#bracket_notation) | слева направо | … [ … ] |
| [new](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/new) (со списком аргументов) | не определено | new … ( … ) |
| [Вызов функции](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Functions) | слева направо | … ( … ) |
| 18 | [new](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/new) (без списка аргументов) | справа налево | new … |
| 17 | [Постфиксный инкремент (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators) | не определено | … ++ |
| [Постфиксный декремент (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators) | … -- |
| 16 | [Логическое отрицание (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#logical_not) | справа налево | ! … |
| [Побитовое отрицание (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#bitwise_not) | ~ … |
| [Унарный плюс (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#unary_plus) | + … |
| [Унарный минус (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#unary_negation) | - … |
| [Префиксный инкремент (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#increment) | ++ … |
| [Префиксный декремент (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#decrement) | -- … |
| [typeof](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/typeof) | typeof … |
| [void](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/void) | void … |
| [delete](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/delete) | delete … |
| [await](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/await) | await … |
| 15 | [Возведение в степень (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#exponentiation) | справа налево | … \*\* … |
| 14 | [Умножение (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#multiplication) | слева направо | … \* … |
| [Деление (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#division) | … / … |
| [Остаток (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#remainder) | … % … |
| 13 | [Сложение (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#addition) | слева направо | … + … |
| [Вычитание (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#subtraction) | … - … |
| 12 | [Побитовый сдвиг влево](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators) | слева направо | … << … |
| [Побитовый сдвиг вправо (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators) | … >> … |
| [Сдвиг вправо с заполнением нулей (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators) | … >>> … |
| 11 | [Меньше (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#less_than_operator) | слева направо | … < … |
| [Меньше или равно (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#less_than__or_equal_operator) | … <= … |
| [Больше (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#greater_than_operator) | … > … |
| [Больше или равно (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#greater_than_or_equal_operator) | … >= … |
| [in](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/in) | … in … |
| [instanceof](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/instanceof) | … instanceof … |
| 10 | [Равно (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#equality) | слева направо | … == … |
| [Не равно (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#inequality) | … != … |
| [Строго равно (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#identity) | … === … |
| [Строго не равно (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#nonidentity) | … !== … |
| 9 | [Побитовое «И» (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#bitwise_and) | слева направо | … & … |
| 8 | [Побитовое исключающее «ИЛИ» (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#bitwise_xor) | слева направо | … ^ … |
| 7 | [Побитовое «ИЛИ» (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#bitwise_or) | слева направо | … | … |
| 6 | [Логическое «И» (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#logical_and) | слева направо | … && … |
| 5 | [Логическое «ИЛИ» (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#logical_or) | слева направо | … || … |
| 4 | [Условный](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Conditional_Operator) | справа налево | … ? … : … |
| 3 | [Присваивание (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#Assignment_operators) | справа налево | … = … |
| … += … |
| … -= … |
| … \*\*= … |
| … \*= … |
| … /= … |
| … %= … |
| … <<= … |
| … >>= … |
| … >>>= … |
| … &= … |
| … ^= … |
| … |= … |
| 2 | [yield](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/yield) | справа налево | yield … |
| [yield\*](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/yield*) | yield\* … |
| 1 | [Запятая / Последовательность](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Comma_Operator) | слева направо | … , … |

Побитовые операторы <https://learn.javascript.ru/bitwise-operators>

**10. Система контроля версий Git и сервис GitHub**

<https://git-scm.com/download/win>

Документация по Git <https://git-scm.com/book/ru/v2/Введение-О-системе-контроля-версий>

Обучение Git <https://githowto.com/ru>

Интерактивное обучение Git <https://learngitbranching.js.org/?locale=ru_RU>

После установки Git идем в терминал, в папку проекта (можно прямо в терминале VSCode, но сначала перезапустить его). Инициализируем пустой локальный репозиторий Git для данного проекта:

… \Project\_1\_Udemy> git init

В проекте появилась папка .git. Конфигурируем пользователя

…\Project\_1\_Udemy> git config --local user.name "RustemV"

…\Project\_1\_Udemy> git config --local user.email rustemv@gmail.com

(для всех проектов –global. . При указании параметра --global используется файл ~/.gitconfig , он хранит настройки конкретного пользователя. В системах семейства Windows Git ищет файл .gitconfig в каталоге $HOME (C:\Documents and Settings\$USER или C:\Users\$USER для большинства пользователей).

Три состояния файлов локального репозитория:

…\Project\_1\_Udemy> git status

- файлы созданы

- файлы индексируются (tracked)

- созданы контрольные точки (commits)

Добавить файлы проекта в локальный репозиторий git

…\Project\_1\_Udemy> git add -A

(-A или --all)

Эту команду выполнять каждый раз при создании нового файла. Или персонально файл:

…\Project\_1\_Udemy> git add main.css

Создание контрольной точки

…\Project\_1\_Udemy> git commit –a –m”текст\_комментария”

(Петриченко рекомендует перед созданием контрольной точки каждый раз выполнять git add -A «Для правильной работы системы контроля версий каждый раз нужно добавлять файлы в индекс»)

Просмотр логов с контрольными точками

…\Project\_1\_Udemy> git log

Выход из лога q

Сервис GitHub

<https://github.com/>

Заводим аккаунт [Rv-csi@yandex.ru](mailto:Rv-csi@yandex.ru) / @Github1812#

Создаем новый удалённый репозиторий (кнопка New). По инструкции на странице привязываем локальный репозиторий к удаленному: в локальном терминале

…\Project\_1\_Udemy> git remote add origin https://github.com/RustemV/Node.js\_Project1.git

…\Project\_1\_Udemy> git branch -M main

…\Project\_1\_Udemy> git push -u origin main

Теперь каждый раз после создания очередной точки

…\ Project\_1\_Udemy > git commit –a –m”текст\_комментария”

отправляем её в удаленный репозиторий на Github

…\ Project\_1\_Udemy > git push

**11. Работа с GitHub с разных компьютеров**

а) Клонирование проекта

Допустим, проект был создан на компьютере № 1.

На компьютере № 2, где еще нет проекта, необходимо клонировать репозиторий. Для этого в Github нужно скопировать путь к нему типа «https://github.com/RustemV/Project\_1\_Udemy.git»



Затем в терминале VSCode:

- перейти в рабочую папку

- выполнить

…\Work> git clone скопированный\_путь имя\_папки\_проекта

Например,

…\Work> git clone https://github.com/RustemV/Project\_1\_Udemy.git Project\_1

На компьютере № 2 создается папка Project\_1 со всеми файлами проекта в ней.

При этом локальный репозиторий уже связан с удаленным. Привязывать не нужно.

После изменения проекта на этом компе № 2 создаем commit и отправляем на Github

…\ Project\_1\_Udemy > git add -A

…\ Project\_1\_Udemy > git commit –a –m”modified at PC#2”

…\ Project\_1\_Udemy > git push

Теперь, когда нужно работать с проектом на компьютере № 1 выполняем в терминале в папке проекта

…\ Project\_1\_Udemy > git pull

А затем, после модификации проекта, обычным образом создается контрольная точка

…\Work > git add -A

…\Work > git commit –a –m”modified at PC#1”

…\Work > git push

Если к моменту выполнения git push , в репозитории на Github’е произошли изменения, то будет выведено предупреждение

To https://github.com/RustemV/Project\_1\_Udemy.git

! [rejected] main -> main (fetch first)

error: failed to push some refs to 'https://github.com/RustemV/Project\_1\_Udemy.git'

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

hint: See the 'Note about fast-forwards' in 'git push --help' for details.

с предложением предварительно выполнить интеграцию удаленных изменений с помощью git pull . При выполнении git pull произойдет merge – слияние изменений внесенных в локальном репозитории с изменениями в удаленном репозитории. Ввести комметарий, Ctrl+C 2 раза, :wq!

В списке коммитов на Github появится Merge.

Игнорирование

Чтобы Git не помещал некоторые файлы, лежащие в проекте, в репозиторий, в папку проекта нужно положить файл .gitignore, в котором указываются игнорируемые файлы.

Пример .gitignore

# See https://help.github.com/articles/ignoring-files/ for more about ignoring files.

# dependencies

/node\_modules

/.pnp

.pnp.js

# testing

/coverage

# production

/build

# misc

.DS\_Store

.env.local

.env.development.local

.env.test.local

.env.production.local

npm-debug.log\*

yarn-debug.log\*

yarn-error.log\*

См. репозиторий <https://github.com/github/gitignore>

Или генерируйте удобные .gitignore файлы для вашего проекта на <https://www.toptal.com/developers/gitignore>

GitKraken

Программа для управления коммитами <https://www.gitkraken.com/>

**12. Практика, ч.1. Начинаем создавать приложение**

Задание

/\* Задание на урок: \*/

'use strict';

/\*1) Создать переменную numberOfFilms и в неё поместить ответ от пользователя на вопрос:

'Сколько фильмов вы уже посмотрели?'\*/

let numberOfFilms;

numberOfFilms = +prompt('Сколько фильмов Вы уже посмотрели?', '');

console.log(numberOfFilms);

/\*2) Создать объект personalMovieDB и в него поместить такие свойства:

    - count - сюда передается ответ на первый вопрос

    - movies - в это свойство поместить пустой объект

    - actors - тоже поместить пустой объект

    - genres - сюда поместить пустой массив

    - privat - в это свойство поместить boolean(логическое) значение false \*/

let personalMovieDB = {

    count: numberOfFilms,

    movies: {},

    actors: {},

    genres: [],

    privat: false

};

/\*3) Задайте пользователю по два раза вопросы:

    - 'Один из последних просмотренных фильмов?'

    - 'На сколько оцените его?'

Ответы стоит поместить в отдельные переменные

Записать ответы в объект movies в формате:

    movies: {

        'logan': '8.1'

    }

Проверить, чтобы все работало без ошибок в консоли \*/

let a = prompt('Один из последних просмотренных фильмов?', '');

let b = prompt('На сколько оцените его?', '');

personalMovieDB.movies[a] = b;

a = prompt('Один из последних просмотренных фильмов?', '');

b = prompt('На сколько оцените его?', '');

personalMovieDB.movies[a] = b;

console.log(personalMovieDB);

При обращении к объекту moves использован синтаксис movies[a], а не movies.a. Такое обращение работает более надежно в случае ввода в качестве значения переменной a произвольного текста с пробелами.

**13. Условия**

<https://developer.mozilla.org/ru/docs/Learn/JavaScript/Building_blocks/conditionals>



Оператор if

if (a == b) {

    console.log('OK!');

} else {

    console.log('Error')

}

Вложенный if

if (a == b) {

    console.log('OK!');

} else if (a > c) {

    console.log('Error 1')

} else {

    console.log('Error 2')

}

Запись if с помощью тернарного оператора

(a == b) ? console.log('OK!') : console.log('Error');

Проверка на строгое соответствие с помощью switch

switch(a) {

    case 49:

        console.log('Меньше');

        break;

    case 50:

        console.log('Равно');

        break;

    case 51:

        console.log('Больше');

        break;

    default:

        console.log('Вне границ');

        break;

}

**14. Циклы**

<https://learn.javascript.ru/while-for>



While

let n = 50;

while (n <= 55) {

    console.log(n);

    n++;

}

Do … while

n = 50;

do {

    console.log(n);

    n++;

}

while ( n <= 55);

For

let num = 51;

for (let i = 1; i<=8; i++) {

    console.log(num);

    num++;

}

Прерывание цикла break

let num = 51;

for (let i = 1; i<0; i++) {

    if (i === 10 ){

        break;

    }

    console.log(num);

    num++;

}

Прерывание цикла continue. Contrinue прерывает только одну итерацию и переходит к следующей.

let num = 51;

for (let i = 1; i<0; i++) {

    if (i === 10 ){

        continue;

    }

    console.log(num);

    num++;

}

**15. Практика, ч2. Применяем условия и циклы**

Задание

/\* Задание на урок: \*/

"use strict";

let personalMovieDB = {

    count: 5,

    movies: {},

    actors: {},

    genres: [],

    privat: false

};

/\* 1) Автоматизировать вопросы пользователю про фильмы при помощи цикла  \*/

let a, b;

for (let i = 1; i <= 2; i++) {

    a = prompt('Один из последних просмотренных фильмов?', '');

    b = prompt('На сколько оцените его?', '');

    personalMovieDB.movies[a] = b;

}

console.log(personalMovieDB);

/\* 2) Сделать так, чтобы пользователь не мог оставить ответ в виде пустой строки,

отменить ответ или ввести название фильма длинее, чем 50 символов. Если это происходит -

возвращаем пользователя к вопросам опять \*/

let aa, bb;

console.log(aa);

for (let i = 1; i <= 2; i++) {

    do {

        //aa = '';

        aa = prompt('Один из последних просмотренных фильмов?', '');

    } while ((aa == '') || aa == null || (aa.length > 50));

    do {

        //bb = '';

        bb = prompt('На сколько оцените его?', '');

    } while (bb == '' || bb == null)

    personalMovieDB.movies[aa] = bb;

}

console.log(personalMovieDB);

/\* 3) При помощи условий проверить  personalMovieDB.count, и если он меньше 10 - вывести сообщение

"Просмотрено довольно мало фильмов", если от 10 до 30 - "Вы классический зритель", а если больше -

"Вы киноман". А если не подошло ни к одному варианту - "Произошла ошибка" \*/

let personalMovieDB.count = +prompt('Сколько фильмов Вы уже посмотрели?', '');

let cnt = personalMovieDB.count;

if (cnt < 10) {

    alert('мало');

} else if (cnt > 30) {

    alert('много');

} else if (cnt >=10 && cnt <=30) {

    alert('классика');

} else {

    alert('ошибка');

}

/\* 4) Потренироваться и переписать цикл еще двумя способами \*/

// 4.1)

let aa, bb;

let i = 0;

do {

    do {

        //aa = '';

        aa = prompt('Один из последних просмотренных фильмов?', '');

    } while ((aa == '') || aa == null || (aa.length > 50));

    do {

        //bb = '';

        bb = prompt('На сколько оцените его?', '');

    } while (bb == '' || bb == null)

    personalMovieDB.movies[aa] = bb;

    i++;

} while (i < 2);

console.log(personalMovieDB);

// 4.2)

let aa, bb;

let i = 0;

while (i < 2) {

    do {

        //aa = '';

        aa = prompt('Один из последних просмотренных фильмов?', '');

    } while ((aa == '') || aa == null || (aa.length > 50));

    do {

        //bb = '';

        bb = prompt('На сколько оцените его?', '');

    } while (bb == '' || bb == null)

    personalMovieDB.movies[aa] = bb;

    i++;

}

console.log(personalMovieDB);

// Код возьмите из предыдущего домашнего задания

Вариант преподавателя

Начало формы

Конец формы

Начало формы

Конец формы

|  |  |
| --- | --- |
|  | /\* Задание на урок: |
|  |  |
|  | 1) Автоматизировать вопросы пользователю про фильмы при помощи цикла |
|  |  |
|  | 2) Сделать так, чтобы пользователь не мог оставить ответ в виде пустой строки, |
|  | отменить ответ или ввести название фильма длинее, чем 50 символов. Если это происходит - |
|  | возвращаем пользователя к вопросам опять. (К любой строке можно обратиться как |
|  | str.length - и получить её длину) |
|  |  |
|  | 3) При помощи условий проверить personalMovieDB.count, и если он меньше 10 - вывести сообщение |
|  | "Просмотрено довольно мало фильмов", если от 10 до 30 - "Вы классический зритель", а если больше - |
|  | "Вы киноман". А если не подошло ни к одному варианту - "Произошла ошибка" |
|  |  |
|  | 4) Потренироваться и переписать цикл еще двумя способами\*/ |
|  |  |
|  | 'use strict'; |
|  |  |
|  | const numberOfFilms = +prompt('Сколько фильмов вы уже посмотрели?', ''); |
|  |  |
|  | const personalMovieDB = { |
|  | count: numberOfFilms, |
|  | movies: {}, |
|  | actors: {}, |
|  | genres: [], |
|  | privat: false |
|  | }; |
|  |  |
|  | for (let i = 0; i < 2; i++) { |
|  | const a = prompt('Один из последних просмотренных фильмов?', ''), |
|  | b = prompt('На сколько оцените его?', ''); |
|  |  |
|  | if (a != null && b != null && a != '' && b != '' && a.length < 50) { |
|  | personalMovieDB.movies[a] = b; |
|  | console.log('done'); |
|  | } else { |
|  | console.log('error'); |
|  | i--; |
|  | } |
|  | } |
|  |  |
|  | if (personalMovieDB.count < 10) { |
|  | console.log("Просмотрено довольно мало фильмов"); |
|  | } else if (personalMovieDB.count >= 10 && personalMovieDB.count < 30) { |
|  | console.log("Вы классический зритель"); |
|  | } else if (personalMovieDB.count >= 30) { |
|  | console.log("Вы киноман"); |
|  | } else { |
|  | console.log("Произошла ошибка"); |
|  | } |
|  |  |
|  | console.log(personalMovieDB); |

**16. Функции**

Несколько подходов:



16.1 Functioon Declaration

function *имя\_функции*(*аргумент1*, *аргумент2*) {

тело функции

return *возвращаемое\_значение*

}

*имя\_функции*(*a, b);*

return останавливает выполнение функции и возвращает результат на уровень выше. Код после return не выполняется.

Переменные, объявленные внутри функции – локальные, существуют только внутри функции, не видны вне функции. Переменные, не объявленные внутри функции, глобальные, также могут использоваться внутри функции. Поиск переменных на уровне выше и далее на следующем уровне и т.д. называется замыканием функции. <https://medium.com/nuances-of-programming/я-никогда-не-понимал-замыкания-в-javascript-часть-первая-3c3f02041970>

<https://learn.javascript.ru/closures>

*имя\_функции* принято начинать с глагола: showForm, getName.

Functioon Declaration создается интерпретатором до выполнения кода. Поэтому её можно вызвать выше, чем она объявлена. Аналогично переменным var.

function calc(a, b) {

    return (a + b);

console.log(‘end’); //unreachable code

}

console.log(calc(4, 6));

console.log(calc(7, 56));

16.2 Function Expression – функциональное выражение

Объявление функции в контексте какого-либо выражения, например присваивания

const *имя\_функции* = function (*аргумент\_1, аргумент\_2*) {

тело функции

return *возвращаемое\_значение*;

};

Сравнение с Functioon Declaration

//Function Declaration

function sum(a, b) {

    return a + b;

}

let s = sum(2, 4);

//Function Expression

const sum = function (a, b) {

    return a + b;

};

s = sum(2, 4);

16.3 Стрелочные функции

Введены стандартом ES6.

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/Arrow_functions>

const *имя\_функции* = (*аргумент\_1, аргумент\_2*) => {

тело функции

return *возвращаемое\_значение*;

}

Если в стрелочной функции одна строка, то можно опустить фигурные скобки – см. код выше, что сильно экономит объем.

//Стрелочная функция

const sum = (a, b) => a + b;

s = sum(2, 4);

console.log(s);

Такая функция не имеет контекста вызова, поэтому их используют для обработчиков событий.

**17. Методы и свойства строк и чисел**

**17.1 Строки**

Посмотреть весь список можно в контекстном меню редактора



Или в документации <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String>

<https://learn.javascript.ru/string>

Также в консоли браузера:

Console.dir(String)

Свойство length – длина строки

let str = "Hello world!";

console.log(str.length);  // 12

\* также пименимо к массивам

Получение символа (readonly)

let str = "Hello world!";

console.log(str[6]);  // w

Метод приведения к верхнему и нижнему регистру

let str = "Hello world!";

console.log(str.toUpperCase());  // HELLO WORLD!

console.log(str.toLowerCase());  // hello world!

console.log(str);               // Hello world!

Сама строка не меняется

Метод поиска подстроки

let str = "Hello world!";

console.log(str.indexOf('ello'));  // 1

Может использоваться как проверка вхождения подстроки в строку. При отсутствии подстроки возвращается -1.

Метод извлечения подстроки

let str = "Hello world!";

console.log(str.slice(6, 11));  // world

возвращает с 6-го до 11-го (не включая!) символа.

При отрицательных аргументах отсчет справа.

При отсутствии 2-го аргумента подстрока извлекается до конца строки.

Метод substring

То же что и slice, но не поддерживает отрицательные аргументы (воспринимаются как 0).

Метод substr

Вырезает подстроку с аргумент\_1 символа длиной аргумент\_2 символов.

let str = "Hello world!";

console.log(str.substring(6, 11));  // world

let str = "Hello world!";

console.log(str.substr(6, 5));  // world

**17.2 Числа**

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number>

<https://learn.javascript.ru/number>

Есть минибиблиотека в браузере Math.

Можно увидеть её состав, если в консоли браузера набрать

>Math.



Округление чисел

const num = 12.2;

console.log(Math.round(num)); // 12

Извлечение чисел

const text = "12.2px";

console.log(parseInt(text)); // 12

console.log(parseFloat(text)); // 12.2

**18. Практика , ч3. Используем функции**

**19. Callback- функции**

Асинхронное выполнение

'use strict';

function first() {

    setTimeout(function() {

        console.log('1st');

    }, 500);

}

function second() {

    console.log('2nd');

}

first();

second();

Вывод будет:

2nd

1st

т.к. в первой функции введена задержка, несмотря на то, что вызов функции first был первым.

Для того, чтобы функция была выполнена после того, как будет выполнена первая, используют callback-функции. <https://learn.javascript.ru/callbacks>

Сallback - это функция, переданная в другую функцию в качестве аргумента, которая затем вызывается по завершению какого-либо действия. Как правило, они используются для асинхронного кода, чтобы быть уверенным, что функция не сработает, пока не закончит свою работу другая функция.

// объявление функции

function *имя\_функции* (*аргумент\_1*, *имя\_callback-функции*) { //см. примечание[[1]](#footnote-1)

тело функции;

*имя\_callback-функции* (); // вызов callback-функции в объявлении функции

}

// объявление callback-функции

function *сallbckFunc() {*

тело callback-функции;

*}*

//вызов функции

*имя\_функции* (*arg\_example*, *сallbckFunc*);

Передача callback-функции в качестве аргумента при вызове функции может быть в виде анонимной функции (тогда предварительное объявление callback-функции отсутствует):

*имя\_функции* (*arg\_example*, function () {

тело callback-функции;

} );

**20. Объекты. Деструктуризация объектов (ES6)**

<https://javascript.ru/tutorial/object/intro>

Пример объекта

const options = {

    name: 'test',    //свойства

    width: 1024,

    height: 1024,

    colors: {

        border: 'black', // вложенные свойства

        bg: 'red',

    },

}

Ключи – это строки. Поэтому можно встретить и такой синтаксис:

const options = {

    ‘name’: 'test',    //свойства

    ‘width’: 1024,

    ‘height’: 1024,

    ‘colors’: {

        border: 'black', // вложенные свойства

        bg: 'red',

    },

}

**20.1 Свойства**

Обращение к свойству - через точку

console.log(options.name);

Обращение к свойству - через [ ]. Внимание – кавычки!

console.log(options['name']);

и к вложенному свойству

console.log(options['colors']['border']);

(см. ниже Деструктуризация объекта).

Удаление свойства

delete options.name;

Перебор всех свойств объекта for in

<https://learn.javascript.ru/object-for-in>

for (let key in options) {

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

}

И то же для вложенных свойств

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]}`);

    }

}

Вывод:

Свойство name имеет значение: test

Свойство width имеет значение: 1024

Свойство height имеет значение: 1024

Свойство border имеет значение: black

Свойство bg имеет значение: red

Определить размер (количество свойств) объекта:

let counter = 0;

for (let key in options) {

        counter++;

}

**20.2 Методы**

Определить размер (количество свойств) объекта.

Метод keys() возвращает массив ключей объекта. Для определения которого используем length:

console.log(Object.keys(options).length); // 4

Создание метода в объекте:

const options = {

    name: 'test',   //свойства

    width: 1024,

    height: 1024,

    colors: {

        border: 'black',

        bg: 'red',

    },

    makeTest: function() {      // метод

        console.log('test message');

    }

}

Обращение к методу

options.makeTest();

Дескрипторы, геттеры и сеттеры свойств

<https://learn.javascript.ru/descriptors-getters-setters>



**20.3 Деструктуризация объекта**

<https://learn.javascript.ru/destructuring#destrukturizatsiya-obekta>

Для того, чтобы при обращении к свойству не использовать конструкцию

console.log(options['colors']['border']);

используется деструктуризация

const {border, bg} = options.colors;

^ ^ ^

нужные свойства свойство верхнего уровня

Теперь они доступны в виде:

console.log(border); //black

console.log(bg); //red

**21. Массивы и псевдомассивы**

const arr = [2, 4, 6, 8, 10];

Длина массива

arr.length

length всегда равна «индекс последнего элемента» + 1.

Удаление последнего элемента

arr.pop();

Добавление элемента в конец

arr.push(12);

Перебор массива

В цикле

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

    console.log(arr[i]);

}

Или **for… of**

for (let value of arr) {

    console.log(value);

}

(в отличие от for… in вызывает на каждом шаге значение, а не ключ. For… in вызывает на каждом шаге ключ и проходит по свойствам в произвольном порядке, поэтому его не следует использовать для Array).

Или с помощью

**Метода forEach(),**

в который передается callback-функция с тремя параметрами: значение элемента, индекс, и сам массив callbackfn(**value: number**, index: number, array: number[]): void

const arr = [2, 4, 6, 8, 10];

arr.forEach(function(val, i, arr) {

    console.log(`${i}: ${val} входит в массив ${arr}`);

})

Вывод:

0: 2 входит в массив 2,4,6,8,10

1: 4 входит в массив 2,4,6,8,10

2: 6 входит в массив 2,4,6,8,10

3: 8 входит в массив 2,4,6,8,10

4: 10 входит в массив 2,4,6,8,10



**Сортировка arr.sort()**

Без ф-ции сравнения

const arr = [2, 14, 26, 78, 10];

arr.sort();

console.log(arr);

Вывод:

[ 10, 14, 2, 26, 78 ]

- сортировка элементов как строки

С ф-цией сравнения (callback)

const arr = [2, 14, 26, 78, 10];

function compareNum(a, b) {

    return a - b;

}

arr.sort(compareNum);

console.log(arr);

Вывод:

[ 2, 10, 14, 26, 78 ]

- сортировка как числа. См. <http://algolist.ru/sort/quick_sort.php>

**Псевдомассивы** –

массивоподобные структуры, но без стандартных свойств и методов.

[Немного о псевдомассивах](https://habr.com/ru/post/336136/)

**22. Передача по ссылке или по значению, Spread оператор (ES6-ES9)**

В случае создания копии

a = b

для примитивных типов данных происходит передача по значению, а для сложных типов данных (массивы, объекты) – по ссылке.

<https://medium.com/@stasonmars/копирование-объектов-в-javascript-d25c261a7aff>

Для создания полноценной копии объекта/массива (по значению) – 4 способа.

**22.1 Нужно произвести поэлементное присваивание:**

function copyObj(srcObj) {

    let dstObj = {}

    let key;

    for (key in srcObj) {

        dstObj[key] = srcObj[key];

    }

    return dstObj;

}

newMyNumbers = copyObj(myNumbers);

Если элементом будет сложный тип, то этот элемент присвоится по ссылке. Это называется поверхностная копия.

Присваивание по значению на всех уровнях вложенности – глубокое копирование (позже).

**22.2 С помощью метода Object.assign:**

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/assign>

newMyObj = Object.Assign({}, myObj);

- объединение объекта с пустым объектом.

**22.3 Для массивов – с помощью метода slice:**

newMyArr = myArr.slice();

**22.4 C помощью Spread оператора … (для массивов начиная с ES6, для объектов – с ES9)**

Оператор развертывания …

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Spread_syntax>

Копирование массива

const newMyArr = [...myArr];

Копирование объекта

const newMyObj = [...myObj];

**23. Основы ООП, прототипно-ориентированное наследование**

Имеем объект

const soldier = {

    health: 400,

    armor: 100

}

И объект

const john = {

    health: 100

}

Прототипное наследование

Устаревший синтаксис

john.\_\_proto\_\_ = soldier;

Теперь можно обратиься к свойству john.armor

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Object_initializer>

Вместо св-ва \_\_proto\_\_ сейчас нужно использовать методы

Object.create()

Object.getPrototypeOf()

Object.setPrototypeOf()

**24. Практика , ч4. Используем объекты**

**25. Отлавливаем ошибки в своем коде при помощи консоли разработчика. Breakpoints**

**26. Динамическая типизация в JS**

****

**26.1 Преобразование в строку (тип string)**

1) String(…) используется редко

String ( 4 ) = “4” – тип string

2) Конкатенация со строкой, в том числе пустой

5 + “” = “5”

Устаревший пример:



Дает



Пример формирования стиля



**26.2 Преобразование к числу (тип number)**

1) Number(…) используется редко

Number ( “4” ) = 4

2) Унарный плюс

+”5” = 5 - тип number

3) ParseInt(…)

ParseInt( “15px”, 10)

Пример преобразования в число



**26.3 Преобразование к логическому типу (тип boolean)**

1) Все несуществующее, пустое, неопределенное будет преобразовываться к False.

0, ‘’, null, undefined, NaN

Остальное – к True.

Пример



- пока элемента нет, условие не выполняется.



- элемент появился, условие выполняется.

2) явный способ с помощью функции используется редко

 - true

3) с помощью !!

!!”44” – true

**27. Задачи с собеседований на понимание основ**

* Какое будет выведено значение: let x = 5; alert( x++ ); ?

Ответ: 5

Комментарий: постфиксный инкремент сначала возвращает переменную, а затем инкрементирует, в отличие от ++х.

* Чему равно такое выражение: [ ] + false - null + true ?

Ответ: NaN

Комментарий: В таких операциях пустой массив преобразуется в пустую строку

[ ] -> “”

Поэтому [ ] + false = “false”.

Далее “false” – null = NaN как результат выполнения нематематических операций.

Далее NaN + true = NaN.

* Что выведет этот код: let y = 1; let x = y = 2; alert(x); ?

Ответ: 2

* Чему равна сумма [ ] + 1 + 2?

Ответ: “12”  
Комментарий: В таких операциях пустой массив преобразуется в пустую строку. Поэтому всё приводится к типу string.

* Что выведет этот код: alert( "1"[0] )?

Ответ: “1”  
Комментарий: К каждому символу строки можно обратиться по индексу.

* Чему равно 2 && 1 && null && 0 && undefined ?

Ответ: null

Комментарий: Логический оператор не приводит числа и результат к логическому типу.

Операция И равна «лжи», если хотя бы один операнд «Ложь».

«И» спотыкается на лжи». В данном случае на null как на операнде равном «лжи»,

* Есть ли разница между выражениями? !!( a && b ) и (a && b)?

Ответ: эти выражения не равны.

* Что выведет этот код: alert( null || 2 && 3 || 4 ); ?
* a = [1, 2, 3]; b = [1, 2, 3]; Правда ли что a == b ?
* Что выведет этот код: alert( +"Infinity" ); ?
* Верно ли сравнение: "Ёжик" > "яблоко"?
* Чему равно 0 || "" || 2 || undefined || true || falsе ?

**28. Получение элементов со страницы**

DOM – document object model. Существует только внутри браузера.

**По уникальному id**

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

- получаем **экземпляр** элемента.

**По тегу**

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

- получаем HTML-**коллекцию** – псевдомассив. Даже, если такой элемент один. (псевдомассив в отличие от обычного массива не имеет свойств и методов).

[Немного о псевдомассивах](https://habr.com/ru/post/336136/)

Чтобы получить **экземпляр** коллекции:

1 способ – при использовании элемента

console.log(btns[1]);

2 способ – при получении элемента

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

**По имени класса**

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

- получаем HTML-**коллекцию** – псевдомассив.

Более современные методы

**По css-селектору**

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

- получаем Node-list – псевдомассив, но у которого есть метод forEach.

Основных видов селекторов всего несколько:

* \* – любые элементы.
* div – элементы с таким тегом.
* #id – элемент с данным id.
* .class – элементы с таким классом.
* [name="value"] – селекторы на атрибут (см. далее).
* :visited – «псевдоклассы», остальные разные условия на элемент.

Селекторы можно комбинировать, записывая последовательно, без пробела:

* .c1.c2 – элементы одновременно с двумя классами c1 и c2
* a#id.c1.c2:visited – элемент a с данным id, классами c1 и c2, и псевдоклассом visited

См. [Знаете ли вы селекторы?](https://learn.javascript.ru/css-selectors)

Использование метода forEach:

earts.forEach( item => {

    console.log(item);

});

Получение только **одного элемента по css-селектору**

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

- получаем **экземпляр** элемента, который первым удовлетворяет набору селекторов.

Замечание.

При получении элемента можно обращаться не к объекту document а к родительскому блоку элемента. Например, если html:

<div class="wrapper">

            <div class="heart"></div>

            <div class="heart"></div>

            <div class="heart"></div>

</div>

То в JS можно так

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

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

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

**29. Действия с элементами на странице**

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

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

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

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

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

посмотреть свойства каждого объекта

console.dir(box);

**Свойство style (объект style)**

Все свойства объекта style являются inline свойствами. Это не css-стили.

Обращение к этим свойствам

box.style.backgroundColor = 'blue';

box.style.width = '500px';

- синтаксис значений, как в CSS.

Inline-стили перебивают остальные, не важно, что написано в CSS.

btns[1].style.borderRadius = '100%';

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

Для изменения сразу нескольких Inline-стилей используем свойство **cssText**

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

или то же с интерполяцией

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

**Перебор элементов с помощью foreach**, если элементы получены через **querySelectorAll** (!) (псевдомассивы, полученные через getElementsBy… не имеют метода foreach)

hearts.forEach( item => {

    item.style.backgroundColor = 'blue'

});

**Создание элемента в javascript**

const div = document.createElement('div'); //в скобках название тега

Такой элемент будет содержаться только в скрипте.

Создание текстового узла – элемента без оболочки тега

const text = document.createTextNode('просто текст');

- используется редко.

**«Застилизовать» созданный элемент – модификация css-классов, добавление, удаление.** Используется свойство classList.

**Добавление css-класса** элементу методом add (класс уже имеется, описан в css)

div.classList.add('black');

На этом этапе элемент создан и находится только внутри java script, к html он никакого отношения не имеет и на странице его нет.

**Методы помещения созданного элемента в страницу**

Применимость того или иного метода можно проверить на <https://caniuse.com/>

1. Добавление элемента в конец тега

document.body.append(div);

- элемент div добавлен в конец body.

(устаревший метод appendchild( ) ).

При разовом обращении к элементу добавление его на страницу можно совместить с получением элемента-контейнера, в который он добавляется.

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

2. Добавление элемента в начало тега

document.body.prepend(div);

- элемент div добавлен в начало body.

3. Добавление элемента перед и после другого элемента.

hearts[0].before(div);

hearts[0].after(div);

(устаревший метод insertbefore( ) ).

4. Удаление элемента

circles[0].remove();

(устаревший метод removechild( ) ).

5. Замена одного элемента другим

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

- метод применяется к заменя\_\_\_му элементу.

(устаревший метод replacechild( ) ).

**Методы редактирования элементов**

1. Помещение текста в элемент

div.innerHTML('Hello world!');

Или как html-код

div.innerHTML("<h1>Hello world!</h1>");

Или текст и только текст

div.textContent = "Hello";

- используется, когда вставляемое содержимое – заведомо текст (в целях безопасности).

2. Вставка фрагмента html-кода перед или после определенных тегов

div.insertAdjacentHTML("afterbegin", "<h2>Hello!</h2>");

- html-фрагмент вставляется в начало элемента (div).

Другие опции метода:

beforeend – в конце элемента

afterend – после элемента

beforebegin – непосредственно перед

**30. Практика. Задание на отработку действий со страницей**

См проект

**31. События и их обработчики**

Список родных событий Javascript <https://oddler.ru/blog/i63>

Справочник по событиям <https://developer.mozilla.org/ru/docs/Web/Events>

Обработчик события - это функция которая срабатывает, как только событие произошло.

В JS три способа назначит обработчик событию.

* + 1. **Использование html-атрибута в верстке.**

Например для события click обработчик события onClick. В html-коде:

       <button onclick = "alert('Clicked')" id="btn">Нажми меня</button>

* + не используется и не рекомендуется.

1. **Использование свойств DOM-дерева для событий.**

Получаем элемент, на который будем вешать обработчик события

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

btn.onclick = function () {

    alert('Clicked!');

}

- используется редко.

Имеет проблему: если объявить повторно, то первое объявление исчезнет, второе объявление заменит первое, и будет работать только второе.

Например,

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

btn.onclick = function () {

    alert('Clicked!');

};

btn.onclick = function () {

    alert('Second clicked!');

};

Вторая проблема – такой обработчик нельзя удалить.

1. **С помощью методов addEventListener и removeEventListener**

Метод addEventListener «привязывает» к событию callback-функцию, которая и содержит обработку

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

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

    alert('Clicked!!')

});

Здесь можно назначить и два таких обработчика

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

    alert('Clicked!!')

});

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

    alert('Clicked OK!!')

});

* + они сработают оба.

События в JS обрабатываются в порядке очереди. Как только оно наступило, оно помещается в очередь независимо от других событий.

См. также <https://developer.mozilla.org/ru/docs/Web/API/EventTarget/addEventListener>

Информация об объекте и типе события, которое произошло, содержится в специальном объекте event (может иметь произвольное имя, например, часто просто e), который передается первым параметром в callback-функцию:

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

    console.log(e);

});

Event имеет много св-в, в частности:

type – тип события,

target – элемент, на котором произошло событие. С ним можно делать все, что угодно, например, удалить:

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

    e.target.remove();

});

См. также <https://developer.mozilla.org/ru/docs/Web/API/Event>

Для того, чтобы потом удалить обработчик события с помощью метода removeEventListener, нужно его вызывать с той же callback-функцией.

Поэтому её целесообразно выносить в отдельную переменную. Например,

const deleteElement = (e) => {

    e.target.remove();

}

btn.addEventListener('click', deleteElement);

btn.removeEventListener('click', deleteElement);

Или реальный пример:

let i = 0;

const deleteElement = (e) => {

    console.log(e.target);

    i++;

    if (i==1) {

        btn.removeEventListener('click', deleteElement);

    }

}

btn.addEventListener('click', deleteElement);

- после однократного выполнения обработчика он удаляется.

См. также <https://developer.mozilla.org/ru/docs/Web/API/EventTarget/removeEventListener>

**Всплытие событий**

Если имеются два элемента, и один из них является родителем другого, и на них, одновременно на родительский и на дочерний элемент, назначены обработчики события, которые обрабатывают одного и то же действие, то можно увидеть, что событие происходит сначала на дочернем (вложенном) элементе, а затем на родительском и т.д., наверх по иерархии DOM-дерева.

Это видно при использовании свойства e.currentTarget (а не e.target).

**Отмена стандартного поведения браузера**

У объекта-события (см. выше event) используется метод preventDefault( ), который помещается в самое начало обработчика события.

Например, при клике по гиперссылке, отменяется переход по адресу в href, а просто выводится в консоль элемент, на котором произошло событие:

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

link.addEventListener('click', event => {

    event.preventDefault();

    console.log(event.target);

});

**Назначение одного обработчика события совокупности элементов** через перебор forEach

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

btns.forEach( btn => {

    btn.addEventListener('click', функция-обработчик);

});

**Опции события**

У метода addEventListener существует также третий аргумент – это опции события.

target.addEventListener(type, listener[, options]);

где

*type* - строка, представляющая тип обрабатываемого события.

*listener* - объект, который принимает уведомление, когда событие указанного типа произошло. Это должен быть объект, реализующий интерфейс EventListener или просто функция JavaScript.

*options* – необязательный, Объект options, который определяет характеристики объекта, прослушивающего событие. Доступны следующие варианты:

*capture*: Boolean указывает, что события этого типа будут отправлены зарегистрированному обработчику listener перед отправкой на EventTarget, расположенный ниже в дереве DOM.

*once*: Boolean указывает, что обработчик должен быть вызван не более одного раза после добавления. Если true, обработчик автоматически удаляется при вызове.

*passive*: Boolean указывает, что обработчик никогда не вызовет preventDefault(). Если всё же вызов будет произведён, браузер должен игнорировать его и генерировать консольное предупреждение. Пример Улучшение производительности прокрутки с помощью passive true

*mozSystemGroup*: Boolean указывает, что обработчик должен быть добавлен в системную группу. Доступно только в коде, запущенном в XBL или в расширении Chrome.

Применение опции *once*:

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

- обработчик выполнится только один раз. Не придется использовать метод removeEventListener, как выше.

**32. Навигация по DOM - элементам, data-атрибуты, преимущество for/of**

Для доступа к разным уровням документа используем свойства документа.

К body можем обратится через:

console.log(document.body);

К head можем обратится через:

console.log(document.head);

Чтобы выйти на уровень выше, на **родительский уровень**, на html, используем:

console.log(document.documentElement);

Для перехода к **дочернему элементу** используем св-во элемента childNodes

console.log(document.body.childNodes);

* + получаем псевдомассив из дочерних элементов (узлов, Nodes) родителя (здесь – body). Переносы строки – в виде узлов text.

Разница между DOM-элементом и DOM-узлом: каждая сущность документа будет узлом, но не каждый узел элементом.

К первому и последнему ребенку можно перейти посредством

console.log(document.body.firstChild);

console.log(document.body.lastChild);

Получение родительского узла. Например, для элемента с id=”current”:

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

Получение следующего родительского узла

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

Переход к следующему соседу-узлу и к предыдущему

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

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

В качестве селектора можно использовать data-атрибуты. <https://developer.mozilla.org/ru/docs/Learn/HTML/Howto/Use_data_attributes>

Например для html-элемента

<li data-current="3">3</li>

используем

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

Свойства-аналоги для получения элементов, а не Node

parentElement

nextElementSibling

previousElementSibling

firstElementChild

lastElementChild

Для childNodes буквального аналога нет. Используем forEach или, когда нельзя, то for … of

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

    if (node.nodeName == '#text') {

        continue;

    }

    console.log(node);

}

- текстовые node (в частности, переносы строки) имеют свойство nodeName=”#text”. Такие ноды мы пропускаем.

[ Из комментариев к уроку

Альтернатива для свойства childNodes - это свойство children - выдает нам html-коллекцию только из Elements, без ненужных нам текстовых Node]

**33. Практика. Используем события на странице проекта**

Для того, чтобы избежать ошибок, связанных с тем, что js начинает работать с элементами до того, как DOM- структура подготовлена (загрузилась), существют события:

* + - 1. load – происходит, когда страница полностью загрузилась. Его можно ждать достаточно долго;
      2. DOMContentLoaded – происходит, когда завершается построение C, а не загрузка всей страницы.

Для того, чтобы js код начал выполняться, когда DOM-структура будет готова, весь код помещают внутрь коллбэк-функции-обработчика этого события DOMContentLoaded.

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

    js код;

});

**34. События на мобильных устройствах**

Большая статья с примерами <http://youon.ru/Андроид/Разработка/touch-sobytiya-na-javascript-multitach-realizatsiya>

Полезная статья <https://habr.com/ru/company/sibirix/blog/227175/>

**События**

touchstart

touchmove

touchend

touchenter

touchleave

touchcancel

**Свойства объекта события**

event.touches

event.targetTouches

event.changeTouches

См. библиотеки, например, Hammer.js

**35. Async, defer, динамические скрипты**

Подключение скрипта в html:

|  |  |  |
| --- | --- | --- |
| вариант |  | Проблемы |
| 1 | В теге head | * если в скрипте есть взаимодействие с элементами DOM, а они еще не загрузились, то будет ошибка. * на время загрузки скрипта блокируется построение страницы. |
| 2 | В конце тега body | * если верстки очень много, то скрипт будет загружаться после ее построения, с большой задержкой. |

Для решения этих проблем существуют два атрибута:

|  |  |  |
| --- | --- | --- |
| № | Атрибут | Особенности |
| 1 | **defer**  <script defer src="js/script.js">  </script> | * Скрипт загружается в фоновом режиме не останавливая, загрузку DOM-структуры * Скрипт начинает выполняться только после того как DOM-структура полностью готова * Несколько скриптов с defer будут загружаться последовательно в порядке подключения, а не параллельно |
| 2 | **async**  <script async src="js/script.js">  </script> | * Страница не ждет асинхронных скриптов, содержимое страницы просто обрабатывается и отображается * Событие DOMContentLoaded и асинхронные скрипты не ждут друг друга * Скрипты с async загружаются в фоновом режиме и запускаются сразу как были загружены, не ожидая других событий и скриптов. То есть абсолютно независимы.   Используется для скриптов, независящих от DOM-структуры и от других скриптов, например, метрик и счетчиков |

Еще один способ помещения скрипта на страницу:

**Динамические скрипты**

Подключаются через создание DOM-элемента

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

script.src = 'js/test.js';

document.body.append(script);

Подключаемый скрипт начинает загружаться в тот момент, когда он появляется на странице в результате метода append ( ).

По умолчанию динамически подключаемые скрипты ведут себя аналогично скриптам с атрибутом async. Для того, чтобы поведение скрипта изменить на обычное, необходимо изменить свойство async (см. пример).

Практический случай – загрузка с помощью функции:

function loadScript(src) {

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

    script.src = src;

    script.async = false;

    document.body.append(script);

}

loadScript('js/test.js');

loadScript('js/some.js');

test и some будут выполняться именно в таком порядке, друг за другом.

Если убрать script.async = false; то порядок может и нарушиться.

**37. ClassList и делегирование событий**

**37.1 Свойство classList**

<https://developer.mozilla.org/ru/docs/Web/API/Element/classList>

Свойство classList возвращает псевдомассив, содержащий все классы элемента. И имеет следующие свойства и методы:

|  |  |
| --- | --- |
| element.classList.length | Возвращает количество классов у элемента |
| element.classList.item(i) | Метод, возвращающий i-й класс |
| element.classList.add(‘some\_class’) | Добавление класса |
| element.classList. remove(‘some\_class’) | Удаление класса |
| element.classList. toggle(‘some\_class’) | Переключает класс |
| element.classList. contains(‘some\_class’) | True, если класс есть у элемента. False в противном случае. |

**37.2 Делегирование событий**

Для того, чтобы не назначать одинаковые обработчики событий группе однотипных элементов, и чтобы событие обрабатывалось также и на позднее динамически добавленных элементах используют делегирование событий:

обработчик события прикрепляется к элементу, являющемуся родительским для группы однотипных элементов, и при возникновении события отлавливает всплывающее событие.

Динамические элементы добавляются внутрь того же родителя, на котором уже назначен обработчик события.

Дочерний элемент-источник события определяется с помощью event.target.tagName,

например,

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

        console.log('Hello');

    }

или через event.target.matches(*селекторы*)

if (event.target && event.target.matches('button.red')) {

        console.log('Hello');

    }

**38. Создаем табы в новом проекте**

**39. Скрипты и время их выполнения.   
 setTimeout и setInterval**

Для выполнения действий через определенный промежуток времени существуют функции:

|  |  |
| --- | --- |
| setTimeout(*функция*, *таймер*);  Пример:  const timerId = setTimeout(function() {      console.log('Hello');  }, 2000);  Или  const timerId = setTimeout(logger, 2000); | * Однократно выполняет функцию после заданной задержки   timerId - числовой ID, который может быть использован позже c [clearTimeout()](https://developer.mozilla.org/en-US/docs/Web/API/clearTimeout). |
| setInterval (*функция*, *таймер*);  Пример:  const timerId = setInterval(function() {      console.log('Hello');  }, 2000);  Или  const timerId = setInterval(logger, 2000); | * циклически вызывает функцию с фиксированной паузой между каждым вызовами. * не учитывает время выполнения вызываемой функции. Если она выполняется дольше заданного интервала, то задержки перед следующим вызовом функции не будет.   В таких случаях используют рекурсивный setTimeout.   * Обнуляется clearInterval(timerId) |

Рекурсивный setTimeout:

let id = setTimeout(function log() {

*некоторые действия*;

    setTimeout(log, 500);

}, 500);

- по прошествии таймера 500 мс запускается ф-ция log, которая выполняет некоторые действия, а затем включает таймер, по прошествии которого запускается функция log, которая выполняет некоторые действия, а затем включает таймер, по прошествии которого …

Стоит отметить, что пул идентификаторов, используемых setTimeout() и setInterval(), является общим, что означает, что технически вы можете использовать clearTimeout() и clearInterval() взаимозаменяемо. Однако для ясности вам следует избегать этого.

**40. Работа с датами**

JavaScript предоставляет функции-конструкторы для множества встроенных объектов языка: например, Date. Он содержит дату и время, а также предоставляет методы управления ими.

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date>

<https://learn.javascript.ru/date>

|  |  |
| --- | --- |
| const now = new Date() | Возвращает объект-текущую дату (время UTC)  2021-11-29T12:48:35.113Z |
| const now = new Date(‘2021-11-29’) | Возвращает объект даты на основе строки  2021-11-29T00:00:00.000Z  (см. ниже метод Date.parse) |
| const now = new Date(2021, 10, 29, 20) | Возвращает объект даты на основе числовых аргументов (месяца нумеруются с нуля: ноябрь это 10)  2021-11-29T17:00:00.000Z |
| const now = new Date(0) | Возвращает объект даты на основе числового аргумента – количества миллисекунд от 01.01.1970  1970-01-01T00:00:00.000Z |

Методы

|  |  |
| --- | --- |
| now.getFullYear()  now.getMonth()  now.getDate() | Возвращают  год,  месяц,  день |
| now.getHours()  now.getMinutes()  now.getSeconds() | Часы  Минуты  секунды |
| now.getUTCHours() | По Гринвичу |
| now.getDay() | День недели (число от 0 (Вск) до 6 (Сб)) |
| now.getGetTimezoneOffset() | Разница в часовых поясах относительно UTC, в минутах |
| now.getGetTime() | Количество мс с 01.01.1970 |
| now.setHours() | Устанавливает время (час) (надо использовать в браузере, а не в консоли VSCode) |
| new Date.parse(‘2021-11-29’) | То же, что и  new Date(‘2021-11-29’) |

Измерение времени выполнения кода

let start = new Date();

for (let i=0; i<100000; i++) {

    let some = i\*\*3;

}

let end = new Date();

console.log(end-start);

**41. Создаем таймер обратного отсчета на сайте**

Обратить внимание на то, что в setInterval функция должна передаваться, то есть в качестве аргумента она должна указываться без скобок, иначе это будет вызов функции, а передаваться будет уже результат отработавшей функции, что бессмысленно для setInterval.

Таким образом должен быть следующий синтаксис:

* либо функция без аргументов

function updateClock ( ) {

...

}

setInterval (updateClock, 1000);

* либо, если функция с аргументами, то

function updateClock (endTime) {

...

}

setInterval ( ( ) => updateClock(deadline), 1000);

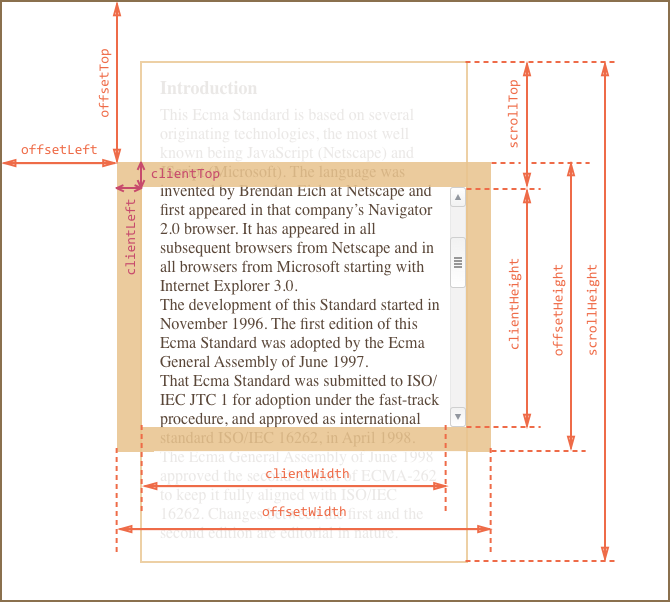
то есть создаем анономную функцию, которая будет вызвана через таймаут и результатом которой будет вызов upDateClock(endtime)

**42. Параметры документа, окна и работа с ними**

Document – объект, который содержит всю html-структуру.

Window – окно [браузера], в котором показывается document.

Screen – это монитор, видимый экран.



Свойства и методы элемента

|  |  |
| --- | --- |
| clientWidth  clientHeight | readonly   * включает padding * не включает border, margin, scroll   То есть, например, если  clientWidth = width + padding - scroll  то свойство css  box-sizing: border-box;  изменяет поведение так:  clientWidth = width - scroll |
| offsetWidth  offsetHeight | readonly   * включает padding * включает border, margin, scroll   То есть совпадает со значениями в css |
| scrollWidth  scrollHeight | readonly  Вся высота/ширина элемента с учетом прокрутки, то есть и невидимая в данный момент часть   * не включает scroll |
| scrollTop  scrollLeft | read-write  Позволяет определить величину пролистанного содержимого элемента, которое не показывается |
| getBoundingClientRect()  Пример:  box.getBoundingClientRect().top  .x, .y,  .top, .bottom,  .left, .right,  .width, .height | Метод элемента возвращающий объект DOMRect, свойствами которого являются координаты минимального по размеру прямоугольника, который заключает в себе элемент, относительно окна браузера с началом координат в верхнем левом углу.   * X/Y-координаты начала прямоугольника относительно окна. * Y-координата верхней/нижней границы прямоугольника. * X-координата левой/правой границы прямоугольника. * ширина/высота прямоугольника. * Меняются при прокрутке (могут быть отрицательными) |

**Computed стили**

Реально примененные стили CSS в результате вычисления браузером.

Read only.

Получение:

**43. Создаем модальное окно**

1. при передаче callback-функции в качестве аргумента скобки не пишутся, т.к. это не вызов функции [↑](#footnote-ref-1)