Плагины

jsHint – выявляет ошибки в коде (опечатки, несостыковки итд)

.jshintrc – файл с настройками по выявлению ошибок в коде для jsHint. Если мы его размещаем в папке, то все папки что лежат с ним рядом (т.е в одной папке с ним) будут иметь такие настройки. Но вот если внутри одной из этих папок будет свой файл .jshintrc, то для этой конкретной папки, где он лежит, он будет перебивать настройки первоначального файла.

Не забыть потом включить **Enable** !!!

**JS**

Js не умеет работать с файлами на компьютере и не умеет работать между вкладками браузера. Мы не можем создать какую-то кнопку на странице чтобы по ее нажатию поменялось что-то на другой странице браузера. Сделано это с целью безопасности.

**Как подключать js**

1. Подключение js прямо в файле html при помощи тега <script>, где мы напрямую пишем код, который нам нужен.
2. Подключение файла js через тег <script src=”js/script.js”>, указав путь до этого файла через ссылку. Если написать код внутри тега, то он будет игнорироваться.

**Скрипт всегда нужно подключать в самом конце, перед закрывающимся тегом </body>**. Если это сделать в начале, то мы просто не сможем ни к чему привязаться, так как верстка еще не сформирована. А также пользователь будет ждать пока подгрузиться js файл и будет видеть пустой экран несколько секунд.

Код можно запустить в консоли VS и посмотреть, как он работает. Выделяем код и запускаем команду Run Code.

**Объявление переменных**

Имя переменной может состоять из букв, цифр, символа доллара и нижнего подчеркивания. Первый символ **никогда** не должен быть цифрой. Также переменные написанные в разном регистре это разные переменные (abc ABC).

let chislo\_1 = 5; - Изменяемая переменная

const chislo\_2= 10; - Постоянная (не изменяемая) переменная. Один раз задали и такое значение и останется. **Прямых констант в JS не бывает.** Тут константы не совсем константы.

Хороший тон использовать const везде, где это возможно.

Разница между let и var.

1. Var name= “Ivan”; - Старый формат записи переменной.

Разница между let и var в том, что переменную let нельзя использовать до объявления. В этом случае будет ошибка. Переменную var можно использовать до объявления. В этом случае ошибки не будет, а переменная будет иметь тип данных underfined. Это может привести к неправильной работе программы.

Использование переменной до ее объявления называется хостинг? (всплытие)

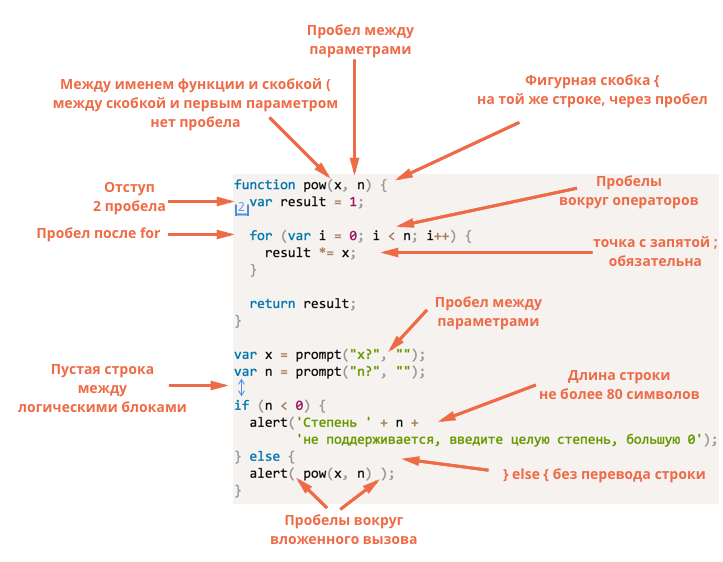
1. Переменные let и const видны в блоке кода ограниченного фигурными скобками. А переменная var не ограничена.
2. Новый стандарт не поддерживается в старых версиях браузера. Сайт Can I Use (<https://caniuse.com/>) показывает какую переменную(и не только) можно использовать в каких браузерах.

Как можно понять, что мы используем современную версию кода

“use strict”; – когда такая строка прописана в js файле, мы говорим нашему файлу, что мы работаем в современном режиме. Где не работают некоторые неточности, которые были в старых стандартах JS. Эту строку нужно прописывать в начале документа, либо в начале функции.

Например, в старом режиме можно было создать переменную без ее объявления A = 15; В режиме use strict этого сделать нельзя, будет ошибка.

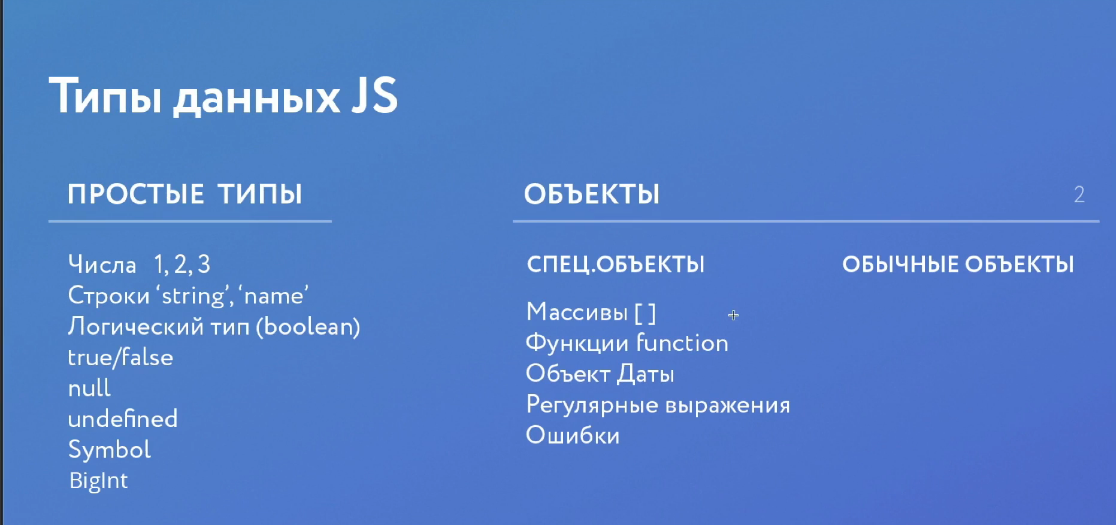
Также чтобы код был понятен нужно стараться использовать стилистические правила.



**; - точка с запятой**

Ставится после объявления переменной или когда закончился какой-то логический блок

**Классификация типов данных**



Тип данных в JS определяется автоматически. **Указывать его явно нельзя**.

**Числа** – мы не можем присвоить число, которое будет больше чем 2 в 53 степени.

Числа могут быть как целыми так и дробными

Let number = 4.7; (для разделителя ипользуем . )

Также можно получить числовые значения, которые относятся к данному типу это инфинити или бесконечность(которое можно получить при деление на ноль) и NaN которое получается если использовать не математическую операцию

**Строки** – в js нет разницы какие кавычки использовать при инициализации строки одинарные или двойные “”, ‘’. Let stroka = “dfg123dfg”;

**Специальное значение Null** – когда чего-то просто не существует. referenceError – ошибка ссылочного типа.

Значение null не относится ни к одному из типов выше, а образует свой отдельный тип, состоящий из единственного значения null:

var age = null;

В JavaScript null не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках. Это просто специальное значение, которое имеет смысл «ничего» или «значение неизвестно».

В частности, код выше говорит о том, что возраст age неизвестен.

**Специальное значение Undefined** – это когда переменная объявлена, но не инициализирована. Она существует, но туда ничего не присвоено.

Значение undefined, как и null, образует свой собственный тип, состоящий из одного этого значения. Оно имеет смысл «значение не присвоено».

Если переменная объявлена, но в неё ничего не записано, то её значение как раз и есть undefined:

var x;

alert( x ); // выведет "undefined"

Можно присвоить undefined и в явном виде, хотя это делается редко:

var x = 123;

x = undefined;

alert( x ); // "undefined"

В явном виде undefined обычно не присваивают, так как это противоречит его смыслу. Для записи в переменную «пустого» или «неизвестного» значения используется null.

**Объект** – это коллекция данных. Структура, которая используется для хранения любых данных.

Const obj = {

Name: “Igor”,

Age: 25

}

Console.log(obj.name); // Igor

**Bigint** – отображает большие числа (больше чем 2 в 53 степени )

**Простое общение с пользователем**

1. Alert(‘Hello’); - вывод текста в браузере.
2. Let ares = confirm(‘Are you here?’); Если мы хотим получить ответ Да\Нет.
3. Let ans = prompt(“Are you 18 ?”, “”); - Если мы хотим получить развернутый ответ. Во вторых кавычках указывается тот ответ что будет в строке по умолчанию.

Type of(имя переменной) – оператор, который проверяет на тип данных.

**Команды alert, confirm, prompt блокируют построение страницы, до того, как они будут выполнены (т.е не выполняется тег <body>)**

**Интерполяция ES6**

Интерполяция значит, что мы прямо внутри строки можем вставить значение переменной.

Когда строка состоит из множества конкатинаций это выглядит не совсем понятно и для этого была придумана интерполяция.

Интерполяция работает **только внутри косых кавычек**(``). Интерполяция вводится через знак $. Например, ${category}

Console.log(`http…../${category}/5`);

**Операторы в JS**

Сложение вычитание деление умножение

Если сложить строку и число, то выйдет строка.

Если складывать пустой массив [] с чем-то, то выйдет строка [] + false // false как строка

Если из строки вычесть null, то будет NaN “asdf” – null = NaN

Инкремент и декремент – увеличивают и уменьшают значения на 1

Переменные можно объявлять через запятую let inc = 10, dec =20;

Префиксные и постфиксные инкремент и дикремент. Отличается тем, что для ++abc такой записи (префиксной) сначала происходит увеличение на 1, а потом работа с этой переменной. А для такой записи (постфиксной) abc++, сначала происходит работа с действующим значением переменной, а потом значение увеличивается.

% - возвращает остаток от деления двух чисел

== - оператор сравнения (true\ false). П.С. число можно сравнить со строкой 8==”8”(true) сравнивается по значению, а не по типу данных. Если мы хотим, чтобы строго число было сравнено со строго числом, то мы должны использовать строгое равенство.

=== - строгое равенство 8 === “8” (false). Сначала идет сравнение по типу данных, потом по значению.

!== - строгое неравенство по типу

&& оператор И.

|| оператор ИЛИ

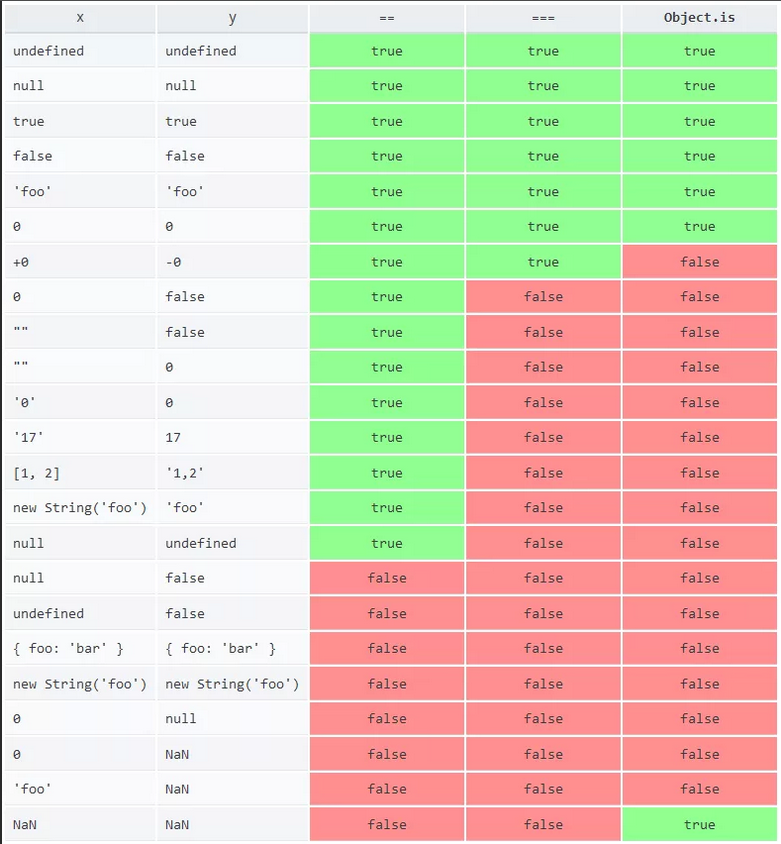
! – оператор НЕ (отрицание)

Таблица главенства операторов - <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Operator_Precedence#table>

Строгое и не строгое сравнение

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

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global\_Objects/undefined



**Git**

После того как мы указали наше имя и email мы можем проверить их в конфигурационном файле. Заходим .git – файл config и там указаны наш имя и email.

Git Kracken – программа для удобства просмотра коммитов и работы с репозиториями.

**Условия**

If (условие){

} else if (условие) {

} else {

}

**Тернарный оператор**

(x == 10) ? console.log(x=10) : console.log(x не равен 10);

**Оператор** **Switch** – проверяет на строгое соответствие .

Switch (num){

Case 49:

Console.log(49);

Break;

Case 50:

Console.log(50);

Break;

default:

Console.log(Ничего не подошло);

Break;

}

**Циклы**

Пока условие выполняется мы выполняем этот цикл.

While (условие){

}

Do{} while(условие) – делай что-то пока условие верно

For(let i= 1; i<10; i++) { - цикл for

If(I == 5)

Break;

}

Если внезапно нужно выйти из цикла по какому-то условия, то можно использовать break.

**Break** – оператор прерывает цикл и выходит из него.

**Continue** – этот оператор переводит выполнение кода к следующей итерации(не прерывая цикл) не позволяя выполнить код после себя.

**Функции, стрелочные функции.**

Имя функции должно быть глаголом с припиской того над чем выполняется действие.

Function showFirstMessage(параметры функции) {

Тело функции;

}

Если переменная объявлена и глобально, и внутри функции с одинаковым именем, то все вызовы переменной с таким именем за пределами функции будут иметь значение глобальной переменной. А внутри функции все вызовы с таким именем будут иметь значение локальной переменной. Например,

Let abc = 20;

Function showFirstMessage(параметры функции) {

Let abc = 10;

Console.log(abc) - выведется 10.

Return – позволяет вернуть из функции какие-то данные(результат работы функции) во внешний код.

Код после слова return не достижим.

}

Console.log(abc) - выведется 20.



**Function declaration** – такие функции существуют еще до того как наш код запуститься. Можем вызвать перед объявлением данной функции. Некоторые разработчики помещают все функции вниз страницы, а сверху их просто используют. В конце функции **НЕ нужно** ставить точку с запятой (;)

Сначала срабатывает html код вместе с css. Потом в конце html кода срабатывает строчка подключения js кода и начинает выполняться он. Первым делом браузер пробегается по js скрипту и находит все переменные var и все function declaration и браузер их объявляет и создает. Только после этого наш код начинает выполняться построчно с самого начала.

**Function expression** – функциональное выражение. Создается только во время выполнения кода построчно. Можно вызвать только после объявления. В конце функции **нужно** ставить точку с запятой (;)

Такую функцию мы присваиваем в переменную и значение переменной это внутреннее состоянии функции.

Чтобы вызвать такую функцию мы должны обратиться к имени переменной – foo();

**Стрелочные функции** – ()=> Такие функции не имеют контекста вызова.

Синтаксис стрелочных функций позволяет сильно сократить код.

Const calc = (a, b) => a + b;

Создаем переменную calc в которую помещаем такую функцию (a, b) => a + b;

Если функция содержит только один аргумент, то можно встретить такую запись:

Const calc = a => a + 10;

Если функция больше, чем в одну строчку то нужно ставить фигурные скобки и писать слово return

Const calc = (a, b) => {

Console.log(a);

Return a + b;

};

**Методы и свойства строк и чисел**

Методы – это вспомогательные функции. Свойства – это вспомогательные значения.

Str.length – это свойство

Str.toUpperCase() – это метод. Вызывается с круглыми скобками

Строку в памяти напрямую изменить нельзя. Если мы делаем над строкой какие-то операции, то у нас появляется совершенно новая строка (с новой областью памяти).

indexOf(‘символ’) – показывает индекс символа в строке

slice(индекс первого элемента, индекс последнего не включая) – обрезает строку. Указывает начало и конец

substring(индекс первого элемента, индекс последнего) - обрезает строку

substr(индекс начала, длинна сколько нужно вырезать символов) - обрезает строку

Math.round(num) – округляет число num до целого

parseInt(строка) – превращает строку в целое число.

parseFloat(строка) – превращает строку в дробное число.

Методы чисел - <https://learn.javascript.ru/number>

Методы строк - <https://learn.javascript.ru/string>

Метод **IsNaN**(переменная) – если внутри переменной не число, то он возвращает true

**Callback функции.**

Некоторые функции срабатывают позже других, даже если были вызваны в коде раньше. Это происходит из-за задержки (ожидание ответа от сервера итд. Мы не можем определить это просто по коду функции будет ли задержка или нет) Это важно учитывать, когда эти функции работают с одними и теми же данными.

Callback функция – это функция, которая должна быть выполнена после того как другая функция завершила свое выполнение. По сути мы должны вызвать внутри одной функции другую, ту, которая должна выполняться после данной внутри которой мы ее вызываем.

Это главный шаблон callback функции. То, что в другую функцию в качестве аргумента в будущем мы сможем передать другую функцию.

Function (lang, callback) {

Тело функции;

Callback();

}

П.С если мы записываем стрелочную колбэк функцию в одну строчку (без скобок{}), то слово return не обязательно писать. Оно опускается. **НО** если мы записываем колбэк в со скобочками или в несколько строчек, то слово return **ОБЯЗАТЕЛЬНО!** Иначе будет возвращать undefined

arr.map((item, index) => item['id'] = index); // [ 0, 1, 2, 3 ] - return опускается

arr.map((item, index) => {item['id'] = index}); //[ undefined, undefined] - так как нет return

arr.map((item, index) => { return item['id'] = index}); //[ 0, 1, 2, 3 ] - return указываем явно

т.е если чо-то собираемся писать внутри {} то оттуда нужен явно return!

**Объекты. Деструктуризация объектов.**

Объекты в js – это ассоциативные массивы. Объекты – это структуры, которое могут сохранять в себе любые типы данных в формате ключ-значение.

Объект может быть создан с помощью фигурных скобок {…} с необязательным списком свойств. Свойство – это пара «ключ: значение», где ключ – это строка (также называемая «именем свойства»), а значение может быть чем угодно.

https://learn.javascript.ru/object

Создание объектов:

1. Вариант

Const obj = {

Name: “Igor”,

Age: 25,

‘like birds’: ‘birds’

}

1. Вариант

Const obj = new Object();

Объекты создаются по типу ключ-значение. Ключ – это свойство (Name), значение –то что там хранится(“Igor”)

Также свойства могут быть добавлены в объект позже путем obj.newField = ‘new’; - добавили новое свойство newField.

**Delete** – оператор, который удаляет свойство

Delete obj.name - удалит свойство name. Не только значение, но и само свойство.

Конструкция для перебора массива и объекта в частности (похоже на foreach в C#)

for ( let key in obj) { – бежим переменной key по объекту obj

console.log(key) – выведет все ключи obj в формате string

console.log(obj[key]) - выведет все значения obj

console.log(obj.key) - выведет ошибку. Возможно из-за того что key содержит строку и для компилятора запись будет выглядеть obj.'name' (это синтаксическая ошибка)

}

Обращаться к значению свойства можно либо через точку(obj.name) либо через [] (obj[‘name’]). Значение в [] должно быть в кавычках.

Свойство может состоять из двух слов. Внутри объекта указывается в кавычках. Тогда к нему можно обращаться только через []. Например, obj[‘like birds’]

**Object.keys(имя объекта**) – метод который берет наш объект и создает массив в котором все элементы это ключи объекта

**Object.keys(имя объекта).length -** количество символов объекта

У объектов есть встроенные методы и свойства, но также в объект мы можем добавлять методы свои.

Const obj = {

Name: “Igor”,

Age: 25,

Colors: {

Bg:red

}

makeTest: function () {тело функции}

}

Obj.makeTest(); - так он вызывается

**Объект, объявленный как константа, может быть изменён**

const user = {

name: "John"

};

user.name = "Pete";

Объявление const защищает от изменений только саму переменную user, а не её содержимое.

Определение const выдаст ошибку только если мы присвоим переменной другое значение: user={…}

П.С. на мой взгляд const дает понять, что переменная ссылается на определнную область памяти и не может быть перенаправлена на другую область памяти. А внутри действующей области памяти можно делать что угодно.

Ограничения на имена свойств объектов

Как мы уже знаем, имя переменной не может совпадать с зарезервированными словами, такими как «for», «let», «return» и т.д.

Но для свойств объекта такого ограничения нет. Нет никаких ограничений к именам свойств. Они могут быть в виде строк или символов. Все другие типы данных будут автоматически преобразованы к строке. Например, если использовать число 0 в качестве ключа, то оно превратится в строку "0"

[Проверка существования свойства, оператор «in»](https://learn.javascript.ru/object" \l "proverka-suschestvovaniya-svoystva-operator-in)

При обращении к свойству, которого нет, возвращается undefined. Это позволяет просто проверить существование свойства.

Также существует специальный оператор "in" для проверки существования свойства в объекте

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

alert( "age" in user ); // true, user.age существует

Но есть особый случай, когда оно не подходит, и нужно использовать "in". Это когда свойство существует, но содержит значение undefined:

let obj = {

test: undefined

};

alert( obj.test ); // выведет undefined, значит свойство не существует?

alert( "test" in obj ); // true, свойство существует!

### [Упорядочение свойств объекта](https://learn.javascript.ru/object" \l "uporyadochenie-svoystv-obekta)

Если мы будем в цикле перебирать все свойства объекта, получим ли мы их в том же порядке, в котором мы их добавляли? Можем ли мы на это рассчитывать?

Короткий ответ: свойства упорядочены особым образом: свойства с целочисленными ключами сортируются по возрастанию, остальные располагаются в порядке создания.

**Деструктуризация объектов** – разделение на более мелкие части.

Const {bg} = obj.colors; - такая структура называется деструкрутризацией

Const {bg} – то что мы хотим вытащить **в качестве отдельных переменных** (т.е bg у нас стало теперь отдельной переменной const к которой мы напрямую можем обращаться)

Имя переменной должно быть точно таким же как и в объекте, если мы захотим вытащить это значение в переменную с другим именем, то она будет indefined. Если нам нужна переменная с другим именем, то проще сделать классическое присвоение. Деструктуризация существует, чтобы долго не лазить к нужной нам переменной в объекте. Const {bg1} = obj.colors; // indefined

obj.colors- откуда мы хотим это вытащить

Как работает JS. Там почти все через объекты. Откуда у строки метод ToUpperCase ? дело в том что строка на базовом уровне тоже является объектом и получает этот метод через цепочку прототипов .

**JS считается объектно-ориентированным языком и все сущности, которые в нем есть сводятся к объектам**

**Массивы и псевдомассивы.**

Массив – это структура, которая содержит элементы по порядку.

Методы массива

arr = [1,2,3,4,5];

arr.pop() – удаляет последний элемент из массива

arr.push(8) – добавляет элемент в конец массива

Удалять или добавить первый элемент у массива, это значить поменять индексы у всех остальных. На больших массивах это может занимать время поэтому классические методы(shift, unsfift) для этого используются редко.

For (let value of arr) – бежим value по arr - Перебор. Такая конструкция работает только с массивоподобными сущностями. Объект такой конструкцией перебрать нельзя (для него существует let value in obj).

Но для массива можно использовать конструкцию let value in arr – тогда выводятся индексы, а не значения(лучше такой вариант не брать)

Если нашему массиву присвоить 100-й элемент arr[99] = 15, то все остальные незаполненные индексы будут пустыми. Если обратиться к такому индексу, то он будет undefined

Также свойство length будет равно – 100 в этом случае.

forEach() – позволяет гибко перебирать все элементы внутри массива. Он похож на конструкцию For (let value of arr), но с некоторыми отличиями. Мы его можем подробнее настроить.

Мы можем добавить внутрь функцию и эту функция будет выполняться каждый раз для каждого элемента массива

Arr. forEach( function(item, I, arr ){} );

Эта колбекфункция может в себя включать 3 аргумента.

1-й аргумент, тот который мы перебираем (например, item). Это значение ключа.

2-й аргумент это номер по порядку (i)

3-й аргумент это ссылка на тот массив который мы перебираем (arr)

**Когда использовать forEach(), а когда (let value of arr) ?**

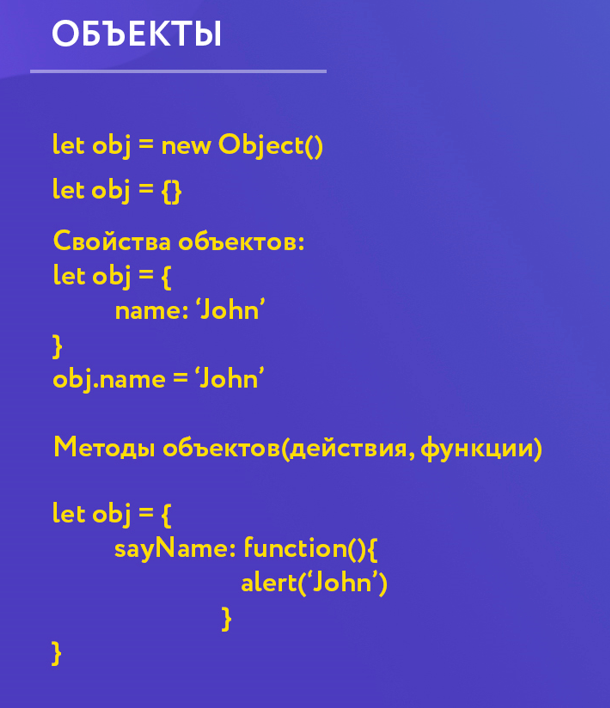
В основном используют forEach(), но в конструкции (let value of arr) можно использовать ключевые слова break, continue. Если они нам нужны то используем (let value of arr)

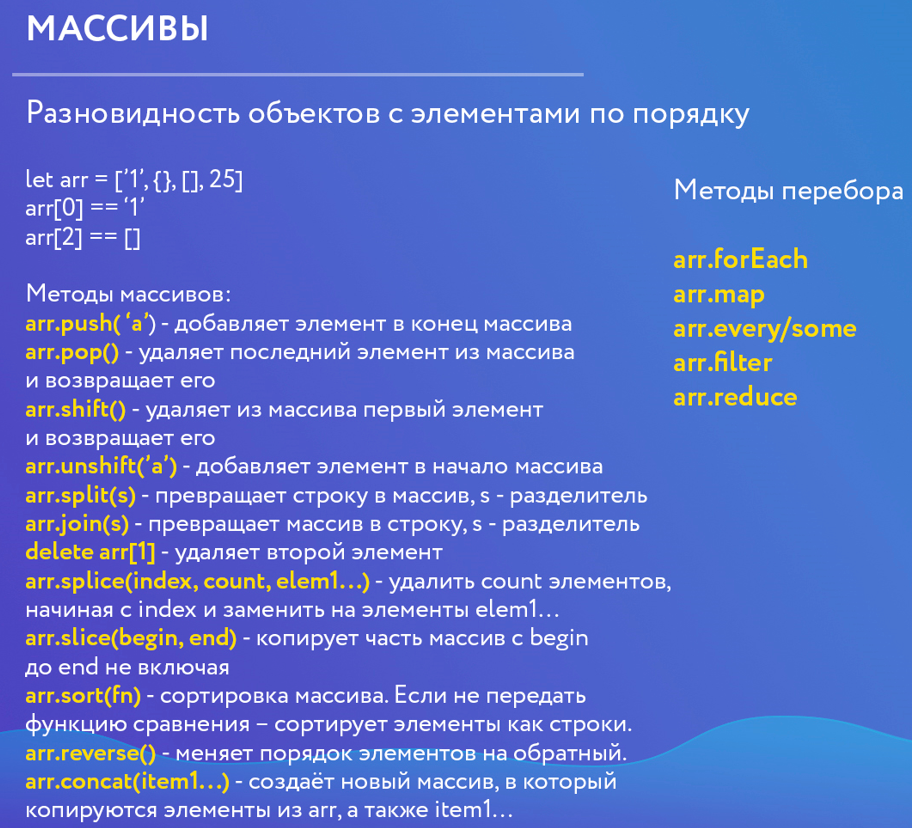
str.split(‘, ’) – разделяет строку str через размелитель (,) на элементы и создает из них массив

arr.join(‘; ’) – склеивает элементы массива в строку через разделитель (;)

arr.sort() – сортирует элементы в алфавитном порядке. Сортирует все как строки. Но этот метод во внутрь может принимать callback функцию

Почитать про sort по ссылке из урока (http://algolist.ru/sort/quick\_sort.php)





Метод forEach просто перебирает массив и ничего с ним не делает, а другие методы могут менять сами элементы (map, filter итд)

Метод [arr.splice(str)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/splice) – это универсальный «швейцарский нож» для работы с массивами. Умеет всё: добавлять, удалять и заменять элементы.

arr.splice(index[, deleteCount, elem1, ..., elemN]) Он начинает с позиции index, удаляет deleteCount элементов и вставляет elem1, ..., elemN на их место. Возвращает массив из удалённых элементов.

Оставшиеся элементы сдвинулись и заняли освободившееся место, а не ключи с значение undefined.

**Псевдомассивы**

**https://habr.com/ru/post/336136/ - о псевдомассивах.**

Псевдомассив - это такой объект структура, которого совпадает со структурой массива. У таких псевдомассивов **не будет никаких методов**. Это просто структура, которая хранит данные по порядку.

Строку можно представить как массив символов и обращаться можно к любому символу

Let str = “asdf”; str[1]// s

Или без объявления переменной “zxcv”[2] // c

**Передача по ссылке или по значению, Spread оператор**

Примитивные типы данных (строки, логические значения, числа) передаются в другую переменную по значению

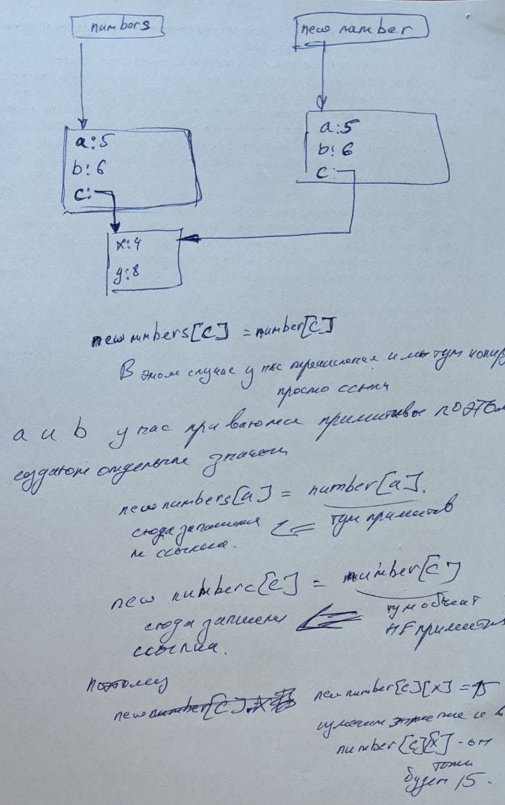
Когда мы работаем с объектами (массивы, функции итд), то они передаются в другую переменную по ссылке. То есть новая переменная начинает ссылаться на тот же объект в памяти и может его менять. В такую переменную кладется ссылка на этот объект, а не сам объект.

Для того чтобы создать действительную копию другого объекта можно сделать следующее:

1. Цикл. Пробежаться по новому объекту и присвоить туда значения из другого.

Глубокие и поверхностные копии объектов.

Поверхностные копии объектов - это объекты которые имеют такие же свойства, что и у объекта откуда мы их брали. И имеют такие же значения, но только для примитивов!!! Если одно из свойств является объектом(любым ссылочным типом), то оно копируется как ссылка



Глубокие копии объектов – это полные копии со всеми примитивными и ссылочными типами.

1. Object.assign(target, … sources) – target- тот объект в который мы хотим поместить sourses

Sourses – объект или объекты(перечисляем через запятую), которые мы помещаем в target

Метод **Object.assign()** используется для копирования значений всех собственных перечисляемых свойств из одного или более исходных объектов в целевой объект. После копирования он возвращает целевой объект.

Метод либо копирует один объект в другой, либо соединяет 2 и более объектов.

**Метод использует поверхностное копирование. Создается поверхностная копия объекта**

1. Для массивов существует метод arr = oldArr.slice(beg, end) позволяет скопировать массив. Массив копируется поверхностно.
2. Оператор разворота(Spread оператор). Разворачивает структуру и превращает ее в набор каких-то данных(отдельных элементов). Развернуть массив можно при помощи синтаксиса const arr = […video, …audio, 45, 20] – в массиве arr будут лежать все элементы из массива video и audio, а также еще 45 и 20. Создается поверхностная копия объекта.

Также спред оператор можно передавать в аргументы метода и он разложится по нужным аргументам function (…video){};

1. Const arr = [1,2];

Const newArr = […arr]; В этом случае создается копия arr и newArr на нее не ссылается видимо потому что в скобках у нас не сама переменная, а уже развернутые элементы, которые с arr никак не связаны.

По такому же принципу можно копировать и объекты.

1. Const obj = [a:1, b:2];

Const newObj = {…obj}; разворачивает все элементы из объекта obj.

**Основы ООП, прототипно-ориентированное**

Прототипно-ориентированное программирование это один из стилей ООП, это частный случай ООП.

**Смысл ООП заключается в том, что мы представляем любую вещь(строку, число итд) как объект с набором свойств, методов.**

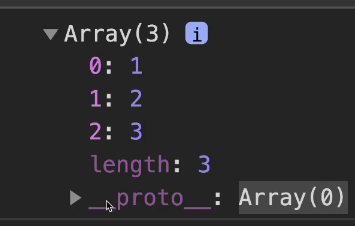
**Строка - это примитивный тип данных. Но как только у строки мы пытаемся вызвать какой-то метод (ToUpperCase), то JS оборачивает эту строку в объект, потом вызывается метод, а после выполнения метода снова возвращает строку.**

Let str = “some”; - тут тип данных у str строка

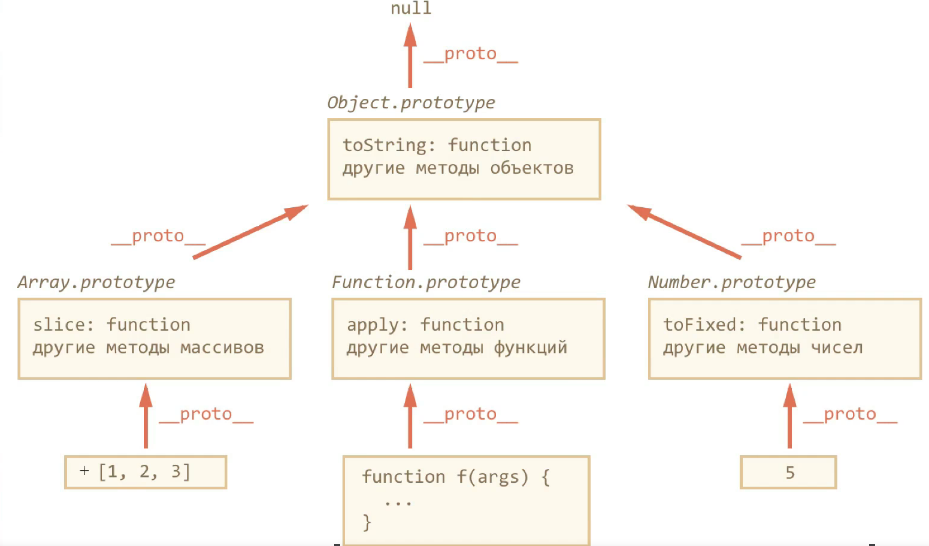
Let strObj = new String(str); - тут тип данных у strObj объект

Когда мы создаем строки или массивы все они могут содержать разное наполнение, но каждый из них обладает одними и теми же методами.

Например, когда мы создаем массив, мы создаем новый экземпляр прототипа массивов (класса массивов)



Сам прототип(класс)Array(0), который нам позволяет создавать отдельные массивы находится в свойстве \_proto или Prototype. В этом свойстве содержатся все свойства и методы которые мы можем применять к массиву. В конце можно увидеть снова \_proto: Object – что показывает от какого прототипа(класса) был унаследован наш прототип Array



Часть свойств что мы можем применить к массиву идут у нас от прототипа Array, а часть свойств от прототипа Object. У объекта уже не существует свойства \_proto так как Object конечное значение.

Как самим создавать прототипы.

Устаревший метод:

Jonh.\_\_proto\_\_= soldier; - старый вариант записи

Jonh теперь наследуется от soldier. У Jonh теперь прототип soldier. И теперь у него есть все свойства и методы soldier.

Современный метод:

Object.setPrototypeOf(кому хотим установить прототип, какой прототип хотим установить)

Object.setPrototypeOf(Jon, soldier)

Let jonh = Object.create(soldier) – создаем новый объект jonh, который будет прототипно наследоваться от soldier

**Prototype в JS это тоже самое что и класс ?**

почти но не совсем

класс - это описание структуры. Оно как бы есть, но пока не создан объект его как бы нет

А proto - это общий предок объектов этого типа. Если одному объекту такого типа в proto что-то прописать - срикошетит на все

**Консоль разработчика, Breakpoints.**

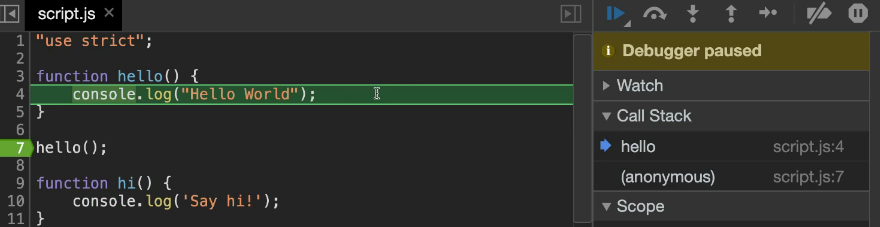
Вкладки:

Element – HTML разметка и стили

Source – исходные материалы которые доступны проекте, который открыт на странице. Тут код из чего состоит ваш сайт.

Network – вкладка отвечает за работу с сетью. Показывает как быстро загружаются разные вещи(скрипты, картинки, html структуры). Также мы можем тут отслеживать запросы, которые будут уходить на сайт(сервер).

CallStack



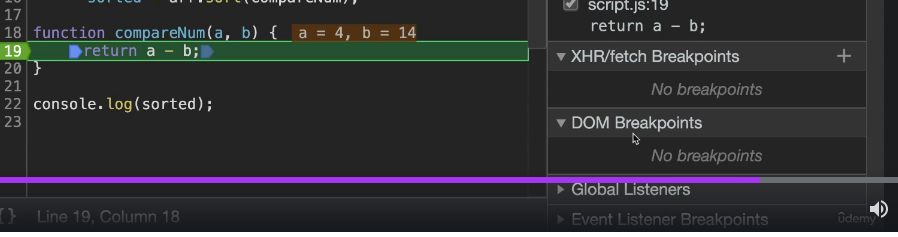
Показывает как мы пришли к той точке где мы сейчас. Если мы внутри функции, то показывает как дошли к этой функции. Из каких других функций наша функция была вызвана. В данном случае первый вызов был на строчке 7. Теперь мы в функции hello на строчке 4. Я так понимаю стек идет снизу вверх.

Scope- переменные с которыми мы сейчас работаем в этой строке.

XHR/fetch breakpoints – вызовы обращения к серверу

Dom breakpoints – можно работать с элементами на странице

Global listener, Event listener – можно работать с обработчиками событий



**Если в коде написать слово debbuger; то когда запустим код, он остановится на этой строчке даже без учета Breakpoints.**

**Динамическая типизация**

Это возможность одного типа данных превращаться в другой. Число может стать строкой итд.

1. Let a = 5;

String(a); - теперь 5 это строка. Мы конвертируем в строку именно значение, а не саму переменную.

console.log(typeof(a)); - это number

console.log(typeof(String(a))); - это string

1. Конкатенация. Когда мы что-то складываем со строкой мы получаем строку.
2. Number(‘55’) - преобразует строку в число.
3. Унарный плюс(+). Преобразует строку в число, если поставить + перед ней. +’25’ – преобразует строку 25 в число.
4. parseInt(‘333’) – преобразует строку в число.
5. 0(ноль), ‘’(пустая строка), null, undefined, NaN – эти значения будут превращаться в **false**, когда переводим их в bool тип.
6. Все остальное будет **true**. Это пустые массивы[], пустые объекты {}, строи итд. Это будет правдой в логическом контексте.
7. Boolean(‘33’) – преобразует строку в логический тип. В данном случае true.
8. !!(‘55’) - преобразует строку в логический тип. В данном случае true.

Матрица сравнений как любой тип данных преобразуется в bool тип <https://dorey.github.io/JavaScript-Equality-Table/>

Из лекции #27.

!!(1 && 2) === (1 && 2) –

console.log( (1 && 2)); - вернет 2

console.log( !!(1 && 2)); - вернет true

&& - запинается на лжи

|| - запинается на правде.

(1 && 2) – когда у нас оба аргумента при логическом сравнении равны(то есть оба true), то выражение вернет последнюю часть. То есть 2. Тип возвращаемых данных бывает разный.

(0 && null) – когда у нас оба аргумента при логическом сравнении равны(то есть оба false), то выражение вернет первую часть. То есть 0. Тип возвращаемых данных бывает разный.

(3 && 0) – когда у нас оба аргумента при логическом сравнении НЕ равны(один true, а другой false), то выражение вернет false часть. То есть 0. Тип возвращаемых данных бывает разный.

(Null || 2&&3 || 4) – это выражение вернет 3. В лекции 27 разобрано

2&&3- вернет 3. Потом

Null || 3 – вернет - 3. Так как правда или лож возвращается правда

3 || 4 – вернут 3. Так как правда или правда вернет первую правду.

A =[1,2,3] B =[1,2,3]

(A == B) – false. Массивы между собой равны, но они ссылаются на разные области памяти, на разные хранилища. Следовательно будет false.

Что выведет этот код: alert( +"Infinity" ); - выведет строку Infinity, но тип данных у нее будет number.

Верно ли сравнение: "Ёжик" > "яблоко"? – строки сравниваются посимвольно. Сравнивается буква Ё и я. У каждого символа есть значения в Unicode. Сравниваются эти значения. Если значение больше, то будет true.

Чему равно 0 || "" || 2 || undefined || true || falsе ? – вернет 2.

**Получение элементов со страницы**

DOM (Document Object Model) – объектная модель документа.

Html код мы можем представить в виде обычного объекта. И как и у любого объекта, у этого объекта тоже могут быть свои методы для того чтобы иметь доступ к определенным элементам на странице.

Чтобы обратиться к DOM нам нужно использовать document и мы увидим весь список свойств и методов который присущ этому глобальному объекту.

Методы для получения элементов со страницы можно разделить на 2 категории. Те что существовали уже давно и современные.

Более давние методы:

Document.geyElementById('nameId') – получить элемент по id. Id должен быть один на странице иначе не метод не поймет к чему обращаться.

Document.getElementByTagName(‘button’) – получим все элементы с тегом button на странице в виде псевдомассива. К нужному элементу можно обратиться по индексу [index]. Даже если на странице всего один элемент с тегом мы получим его в **массиве**.

Document.getElementByClassName(‘nameClass’) – получить элементы по имени класса. Возвращает массив.

Более свежие методы:

Document.querySelectorAll(‘css selector’) – получаем все элементы по данному селектору. Возвращает псевдомассив у которого есть метод forEach

Основные виды селекторов - <https://learn.javascript.ru/css-selectors>

Document.querySelector(‘css selector’) – получаем только один элемент (первый попавшийся)

Можно обращаться не только к document, но и к объекту содержащему элементы.

Const abc = Document.querySelector(‘css selector’) – получаем какие-то элементы из document

Const abc\_1 = abc.querySelector(‘css selector’) – получаем какие-то элементы из abc.

**Действия с элементами на странице**

Console.dir(id) – посмотреть что наш элемент представляет в качестве объекта.

Const box = Document.geyElementById('nameId');

Изменить цвет и ширину. Значения нужно вносить в кавычках.

Box.style.backgroundColor = ‘red’;

Box.style.width = ‘500px’;

Инлайн стили перебивают все остальные. Даже сss стили.

**Стили в VisualStudioCode не выскакивают в качестве подсказки. Их нужно вводить по памяти(width, background итд)**

А так можно назначить сразу несколько стилей в одной строчке

Box.style.cssText = ‘backgroundColor: red; width: ${number}px’;

Произвести над несколькими элементами одно действие.

1. Можно при помощи цикла for
2. При помощи метода forEach (только для querySelector)

Основные методы для работы с элементами страницы:

Создание элемента внутри js скрипта.

Const div = document.createElement(‘div’); - Создание элемента внутри js скрипта. (На странице он не появится???)

Также можно создавать текстовые узлы. Это элементы без оболочки тега.

Const div1 = document.createTextNode(‘какой-то текст’); - этот текст появится на странице.

Можно добавлять (удалять и работать с блоком) классы элементу при помощи метода classList

div.classList.add(‘black’); - добавили класс black к элементу div(**не к тегу, а именно к Const div**)

Добавляем какой-то элемент(например наш div) в конец другого тега.

Document.body.append(‘div’); - обращаемся к блоку body и добавляем внутрь. в конец наш блок div.

Document.querySelector(‘.wrapper’).append(‘div’); - при помощи querySelector(‘.wrapper’) получаем элемент и добавляем в конец этого элемента наш блок div.

appendChild – устаревший метод. Тоже самое что append

Document.querySelector(‘.wrapper’).**prepend**(‘div’) – вставляет элемент в начало другого элемента

Методы before и after

Hearts[1].before(div); - наш div встанет перед элементом Hearts[1].

querySelector(‘.wrapper’).insertBefore(div, Hearts[1]); - устаревший метод. Только в старом коде

querySelector(‘.wrapper’) – обращаемся к родительскому элементу.

div- то что мы вставляем

Hearts[1]- перед этим элементом мы вставляем наш div

Remove – удаляет элемент со страницы

Circle[1].remove();

querySelector(‘.wrapper’).removeChild(Hearts[1]); - устаревшй метод. Обращаемся к родительскому элементу и удаляем элемент Hearts[1]

replaceWith – заменяет один элемент на другой

Hearts[1].replaceWith(circle[1]); - заменяет Hearts[1] на circle[1]

querySelector(‘.wrapper’).replaceChild(чем хотим заменить, что хотим заменить) – устаревший метод. Тольк ов старом коде.

Методы позволяющие редактировать элементы.

innerHTML – позволяет вставлять html структуру внутрь элемента.

div.innerHTML = “<h1>Hello world</h1>”; - внутри элемента div появится другой html элемент, который заменит все внутренние элементы div на данный.

div.innerHTML = “”; - такой код заменит все элементы внутри div на пустоту. У div не будет элементов.

textContent – можно поместить текст внутрь блока

div.textContent = ‘Hello’;

InsertAdjscentHTML – метод позволяющий добавить html структуру

div. InsertAdjscentHTM(‘спец.слово’, ‘<h2>111</h2>’);

div- элемент над которым производим действие

<h2>111</h2> - тот html, который мы хотим вставить

Специальные слова

beforebegin – вставляем <h2>111</h2> перед div

afterbegin – вставляем <h2>111</h2> в начало div. Первым элементов внутри

beforeend - вставляем <h2>111</h2> в конец div. Последним элементов внутри

afterend - вставляем <h2>111</h2> после div

**??? Получается что когда мы делаем const abc = Document.querySelectorAll(‘css selector’); то мы по сути получаем в переменную abc ссылку к этим элементам на странице и можем их менять динамически. Мы получаем не копию этих элементов в переменную abc, а именно ссылку. ???**

**По логике получается так, ведь document это объект. В нем содержатся теги(тоже объекты или массивы элементов) и мы эти ссылочные объекты присваиваем в другую переменную. А при присвоении ссылочного типа в другую переменную туда присваивается только ссылка. Следовательно, когда мы их меняем через нашу переменную, то они меняются и в document.**

Когда мы используем более одних кавычек внутри других, то нам нужно использовать различные виды кавычек. 'url("img/bg.jpg")'; или можно заэкранировать одинаковые кавычки при помощи (\)

'url(\'img/bg.jpg\')'

**События и их обработчики.**

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

https://developer.mozilla.org/ru/docs/Web/API/EventTarget/removeEventListener

Событие – это сигнал от браузера о том, что что-то произошло(клик, двойной клик, наведение мыши, убрать мышь, прокрутка колесика, отправка данных в форме, нажатие клавиш итд. )

Список событий js - <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.

Есть 3 способа назначить обработчик событий:

1. Html атрибут прямо в верстке. На нашем элементе прописываем onclick=”код, который выполнится при срабатывании события” (Используется редко)
2. Использовать свойство DOM дерева для событий. (Используется редко)

Btn.onclick = function() {код1}

Минусы этого варианта:

Если мы забыли(или не знали что он уже есть) и добавили на кнопку другой обработчик события, то он перепишет первый обработчик. Btn.onclick = function() {код2}

Иногда обработчики события нужно удалять и если мы назначаем его при помощи такого синтаксиса(Btn.onclick=fun..), то удалить мы его не сможем.

1. Методами AddEventListener, RemoveEverntListener

Btn. AddEventListener(‘click’, () => {}) – js будет следить за этим элементом, если у нас произошло это событие, то он запустит обработчик

Click- название нашего события

() => {} - колбек функция, которая будет нашим обработчиком. **Первым аргументом всегда будет объект-событие**.

<https://developer.mozilla.org/ru/docs/Web/API/Event> - объект - событие

Если нам нужно передать свои данные, то только начиная со второго параметра. (event, Mydata) => {}

Преимущества

Мы можем назначать несколько разных обработчиков на один и тот же элемент и обработчики в таком случае сработают последовательно, а не перезапишут друг друга.

События в JS выполняются в порядке очереди. Как только новое событие поступило, оно добавляется в очередь независимо от других событий.

Иногда необходимо получать данные об элементе, с которым мы взаимодействуем

Btn. AddEventListener(‘click’, (e) => {})

е – это объект, который описывает, что произошло с элементом.

Важные свойства type(тип события), target(элемент на котором произошло событие)

Удаление обработчика событий

Чтобы иметь возможность удалить функцию из обработчика событий она должна быть вынесена в отдельную переменную.

Const fun = (e) => {код}

Btn. AddEventListener('click', fun); - передавая в обработчик функцию таким образом мы можем потом ее удалить. Так как fun ссылается на конкретную область в памяти. А если бы была просто функция, то на нее не было бы ссылки и метод удаления не знал бы как ее найти.

Btn. removeEventListener('click', fun); - удаляем конкретную функцию fun с обработчика.

Всплытие событий

Дело в том, что если обработчик события привязан к родительскому классу, то он будет срабатывать на всех вложенных элементах. (rod.addEventListener('click', fun);) – будет срабатывать на Кнопка2, Кнопка3. (даже если мы нажимаем на вложенный элемент, то этот элемент все равно находится в области родительского элемента и браузер понимает это так что мы нажали на вложенный элемент и проверить его, а потом он также понимает что мы нажали на родителя и проверит его тоже на события)

                    <div class="rod">

                        <button class="dot">Кнопка2</button>

                        <button class="dot">Кнопка3</button>

                    </div>

Если у нас один элемент вложен в другой и на обоих этих элементах прикреплены обработчики события, которые обрабатывают одно и тоже действие (click в данном случае), то произойдет всплытие событий.

rod.addEventListener('click', fun); - родительский элемент.

dot.addEventListener('click', fun1); - вложенный элемент.

Если нажать на дочерний элемент, то сначала обработчик срабатывает на самом вложенном элементе dot, а затем на родителе rod, если у него тоже есть обработчик. То есть нажав на кнопку мы получим 2 события. (вероятнее всего потому что мы жмем в одно место на экране и браузеру сложно понять какой конкретно элемент мы имели в виду. Поэтому он запускает 2 события)

Если нажать на родительский элемент, то сработает только событие родительского элемента.

**П.С. Когда один элемент вложен в другой и нажимаешь на дочерний, то в target будет указан дочерний элемент.**

Для отмены стандартного поведения браузера.

Стандартное поведение это при нажатии на ссылку переходим по адресу этой ссылки, зажать левую кнопку мыши и вести по тексту, то текст будет выделяться итд.

Event.preventDefault(); -отменяет стандартное поведение для этого элемента. Например, отменяет переход по ссылке

aaa.addEventListener('click', (event) => { Event.preventDefault(); });

Чтобы повесить один и тот же обработчик на разные элементы (например, 10 кнопок), нам нужно их перебрать и на каждый из них повесить обработчик. (или повесить на родителя. это сработает, но то что так нужно делать это уже мои догадки)

Также в обработчик события мы можем добавлять **дополнительные опции**. Это не обязательный параметр и можно его не указывать.

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

rod.addEventListener('click', fun, {once: true});

once: true – говорит нам, что обработчик события сработает всего 1 раз.

**Навигация по DOM элементам**

Взять элемент и обратиться к его родителю или следующему элементу

Document.documentElement – соответствует тегу <html>

Document.head – соответствует тегу <head>

Родительский элемент это тот, который объединяет другие, находящиеся внутри него.

childNodes – метод показывающий дочерние узлы у выбранного элемента

Document.head. childNodes – покажет все внутри head в виде псевдомассива. П.С перенос строки на следующую строчку считается текстовой нодой(Node) или текстовым узлом и мы его увидим в нашем псевдомассиве. В самой верстке мы этого не видим.

Каждая сущность, что находится на странице будет является Dom-узлом(теги, текстовые ноды). Но не каждый Dom-узел будет Dom-элементом (например текстовая нода не является Dom – элементом, а все теги являются Dom-элементом )

Если мы хотим получить только дочерние элементы у выбранного элемента нам придется использовать перебор for(let… of….) {if(что не нужно){continue} что нужно}

firstChild, lastChild firstElementChild, lastElementChild

Document.body.firstChild (lastChild) – позволяют получить первый и последний ноды body

Document.body.firstElementChild (lastElementChild) - позволяют получить первый и последний элементы body

parentNode – получить родительский узел элемента (может быть любая нода, даже текстовая)

parentElement – получить родителский элемент

querySelector(‘.wrapper’).parentNode – получим родителя элемента с классом wrapper

querySelector(‘.wrapper’).parentNode. parentNode – получить родителя родителя нашего элемента

data – атрибуты (прописываются в квадратных скобках [xxx])

document.querySelectorAll('[title]') - получим все элемент с атрибутом title в виде псевдомассива

document.querySelectorAll('[title=”картинка”]') - выведет все элемент с атрибутом title в виде псевдомассива, где значение атрибута будет “картинка”.

nextSibling, previousSibling – получить следующего или предыдущего соседа (Nodes) – если такой метод поставить к обычному тегу мы получим текстовую ноду

nextElementSibling, previousElementSibling – получить предыдущий или следующий элемент(тег)

**Работа с неоптимизированными страницами**

Скрипт может начать выполняться до того момента как верстка будет сформирована. Это происходит, когда различные теги долго подгружаются на странице. На сервер отправляется запрос и ответ с него еще не пришел, а верстка простраивается дальше и начинает выполняться скрипт, а ответ с сервера для некоторых элементов все еще не пришел и эти элементы не подгружены. Следовательно, и скрипт не найдет обращение к этим элементам, и мы получим ошибку. Чтобы этого избежать у нас есть 2 события.

1. Load. Событие срабатывает тогда, когда страница полностью загрузилась. То есть это событие ждет пока загрузятся все элементы на странице(картинки итд). А некоторые из них могут грузиться долго 5-10 секунд. И все это время пользователь ждет.
2. DomContentLoaded. Это событие срабатывает не когда подгружены все элементы, а когда выполнится только построение DOM структуры. Остальные картинки, скрипты, стили будут подгружаться в фоновом режиме и ни на что влиять не будут.

Document.addEventListener(‘DomContentLoaded’);

window.addEventListener(‘DomContentLoaded’); - так тоже можно

**События на мобильных устройствах**

В мобильных браузерах есть табы (когда мы прикасаемся пальцами к сенсору)

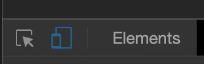
Мобильные браузеры сами по себе нормально отрабатывают со всеми сайтами. И если в десктопной версии мы установили событие клика, оно будет работать в мобильном браузере.

События

1. Touchstart – событие срабатывает при возникновении касания к этому элементу.
2. Touchmove – палец при касании начинает двигаться по элементу
3. Touchend – как только палец оторвался от элемента, то событие сразу выполнилось
4. Touchenter – когда мы ведем пальцем по экрану и натыкаемся на элемент на который повешено это событие. Как только палец зашел за пределы этого элемента.
5. Touchleave - когда мы ведем пальцем по экрану, натыкаемся на элемент и как только мы выходим из этого элемента сразу срабатывает событие.
6. Touchcancel – возникает, когда точка соприкосновения больше не регистрируется на поверхности. Например, на смартфоне мы вышли за пределы браузера. И сработает это событие.

Для мобильных обработчиков рекомендуется всегда отключать поведение браузера по умолчанию для каждого элемента. E. Preventdefault();

Для переключения работы со смартфонами на десктопе нужно включить элемент смартфона



Там мы можем проверять все эти обработчики событий

Так же мы можем узнать сколько сейчас пальцев взаимодействует с элементом через свойство события e.toches. Получим объект touchList, где length – количество пальцев

targetToches – свойство которое показывает сколько пальцев взаимодействует с конкретным элементом

changedToches – список пальцев, которые участвуют в текущем событии

**Загрузка скриптов на страницу**

Когда наш файл (например, index.html) попадает в браузер, наш браузер идентифицирует эту страницу как html и начинает ее загрузку. Подгружается <head> . Из него подгружаются <meta> теги и устанавливаются на странице. Далее тег <title>. Потом подгружаются стили <link>. После <head> браузер переходит к <body> и начинает строить верстку тег за тегом, а в конце подгружает <script>.

Если <script> перенести в тег <head>, то у нас возникнут 2 проблемы. То, что скрипт будет обращаться к элементам, которые внутри <body>, а их еще нет на странице, они не построились. Следовательно, будет ошибка(иногда ошибку не выдает, но скрипт в таком случае выдает неправильные результаты). (П.С. загрузка скрипта и его выполнение походу не одно и тоже. Если мы повесим на скрипт обработчик событий который позволяет выполняться скрипту только после построения дом структуры, то у нас будет следующая модель работы. Сначала в head наш скрипт загружается с сервера и ждет, потом выстраивается вся страница body, потом начинает выполняться скрипт. То есть само событие позволяет подгрузиться скрипту(но не выполнять его), тем самым позволяя тегам идущим за скриптом подгружаться дальше)

Вторая проблема это то, что пользователь будет долго ждать пока загрузится скрипт(зависит от объема скрипта и от сервера на котором лежит скрипт, будет ли он подвисать)

Эти 2 проблемы мы решали тем, что писал тег <script> самым последним в <body>. Но если у нас будет страница, где будет тысяча строк кода перед <script>, тогда пользователю опять придется долго ждать.

**Атрибут defer**

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

Атрибут defer – сообщает браузеру, что он должен продолжать обрабатывать страницу(строить теги html после скрипта) и загружать скрипт в фоновом режиме, а затем запустить этот скрипт, когда он загрузится. Скрипты с атрибутом **defer** всегда выполняются, когда наше DOM дерево готово. Поэтому скрипт даже, если загрузится раньше построения страницы будет ждать пока это построение завершится и только потом запустится. (П.С такие скрипты срабатывают раньше,чем событие DomContentLoaded)

Скрипты с атрибутом defer сохраняют порядок относительно друг друга.

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

<script **defer** src="js/jsForTest123.js"></script>

<p>Hello world</p>

Так как эти скрипты содержат defer они будут загружаться в фоновом режиме, а выполняться будут последовательно. Сначала выполнится весь код из первого скрипта, только потом из второго(тестировал на скриптах с счетчиками до миллиона).

Так соблюдается порядок подключения всех утилит, и мы точно уверены, что страница готова. Можно сначала подключить библиотеку с самыми полезными утилитами. А следующий скрипт тот который использует библиотеку и зависит от нее. При этом вся наша страница не блокируется.

П.С. если скрипты без атрибута defer, то они выполняются последовательно. Правда html теги дальше не грузятся.

**Атрибут async**

1. Страница не ждет асинхронных скриптов и выполняется дальше. Такие скрипты загружаются в фоновом режиме. Запускается сразу, после того как был загружен.
2. Событие DomContentLoaded и async скрипты не ждут друг друга.

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

<script **async** src="js/jsForTest123.js"></script>

Мы не можем знать какой из этих скриптов выполниться первым.

Иногда мы подключаем скрипты(например, метрики и счетчики), которые не особо подвязаны как под DOM структуру, так и под другую функциональность. Такие скрипты могут выполняться сразу.

**Динамические скрипты.** Скрипты, создаваемые внутри js файлов.

По умолчанию ведут себя как async.

const script = document.createElement('script'); - создаем элемент

script.src = "js/test.js"; - заполняем атрибут src

document.body.append(script); - помещаем элемент в конец body. Когда скрипт в котором была создана эта строчка доработает, только потом вызовется данный скрипт, так как в теге body он идет после него.

П.С. если мы создаем такой скрипт внутри скрипта, который расположен в head, то у нас будет ошибка, так как тег body еще не создан. А если такой скрипт внутри body, то все работает.

**ClassList и делегирование событий**

ClassList - это свойство, которое есть у DOM узла. Оно содержит различные методы и другие свойства (например есть свойство, которое покажет нам количество различных классов у элемента)

ClassName - это свойство, которое есть у DOM узла. Оно покажет нам классы, которые есть у элемента в виде строки

Div. ClassList.length; - количество классов у элемента div

Div. ClassList.item(0); - метод позволяющий получить класс, который располагается под определенным индексом. (0) – получим первый класс уэлемента

Div. ClassList.add(‘red’ , ‘color’); - добавляет классы red и color нашему div

Div. ClassList.remove(‘red’); - удаляет класс red нашему div

Div. ClassList.toggle(‘red’); - если red есть на элементе, то метод его удалит. Если его нет на элементе, то метод его добавит.

Div. ClassList.contains(‘red’); - Метод позволяет определять наличие определенного класса(red) на элементе.

**Делегирование событий**

Wrapper – переводится как “обертка”.

Делегирование элементов подходит, когда есть много элементов с одинаковыми обработчиками. Мы вешаем на родителя этих элементов данный обработчик. Мы делегируем событие с родителя на его потомков через определенные условия(например, условие if(target.TagName == ‘BUTTON’)).

Если мы будем использовать обычный перебор(forEach) чтобы навесить на каждый элемент обработчик событий, то это сработает только для элементов на момент вызова этого кода. Если мы потом добавили еще один такой элемент динамически (через js), то на нем обработчик уже не сработает, так как он не был навешан.

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

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

        //код

    }

});

Wrapper – это родитель

event.target – мы проверяем, что мы попали именно в элемент на который можно кликнуть. Дело в том, что не все элементы поддерживают событие – click. Например, тег <br> перенос строки этого не поддерживает, соответственно не имеет такого свойства как target. Чтобы избежать ошибки мы проверяем на event.target

event.target.TagName == "BUTTON" – тут мы проверяем что мы попали именно в кнопку

event.target.mathes(“button.red”) – более широкое условие проверки. Проверяет что мы попали на кнопку(button) у которой есть класс red

Модуль 3.

**Создаем табы в новом проекте**

Стандарт ES6

Если функция вызвана без аргумента, то по умолчанию подставится то, что стоит после =

Function show (arg = 2) {код}; описание функции

Show(); - эта функция вызовется с параметром 2. Так как он указан по умолчанию

Show(3); а эта с параметром 3.

**Управление временем выполнения скриптов**

Очень часто наши функции выполняются не сразу, а через промежуток времени.

setTimeout – это конструкция для того чтобы запустить нашу функцию через определенное время

setTimeout(параметр 1, параметр 2, , параметр 3);

параметр 1 – указывается функция что запустится через время. Принимает объявление функции или ее название. Она не вызывается прямо здесь и сейчас.

параметр 2 – через какое время она вызовется. 2000мс – 2 секунды.

параметр 3 – это параметр, который попадет в параметр параметра1. Hello попадет в параметр text.

const timer = setTimeout(function(text){

    console.log(text);

}, 2000, 'Hello');

clearTimeout(timer) - функция останавливающая наш timer для setTimeout

clearInterval(timer) – функция останавливающая наш timer для setInterval

setInterval – конструкция повторяет нашу функцию через определенное время. Параметры те же.

**Чем рекурсивный setTimeout лучше, чем setInterval ?** Если вдруг функция выполняется дольше, чем мы поставили задержку, то setInterval будет вызывать следующую функцию не дожидаясь пока предыдущая выполниться. Если функция выполняется 5 секунд, а задержка 1 секунда, то пока выполняется первая функция будут запущено еще 4 таких функции. Чтобы этого избежать используют рекурсивную функцию setTimeout. Вызывают ее в конце всего кода. Так мы уверены что функция внутри setTimeout полностью выполнилась, потом идет заданное ожидание и снова выполняется setTimeout.

**Работа с датами**

Дата – это такой специфический объект, который содержит в себе определенные методы и свойства.

Конструктор создания дат new Date();

Const now = new Date(); - при такой конструкции мы в переменную now получаем время на момент выполнения этой строчки кода. То есть сейчас.

Const now = new Date('2022-04-15'); - можно передать конкретную дату.

Const now1 = Date.parse('2022-04-15'); - обрабатывает строку в заданном формате и возвращает таймстамп дает значение в миллисекундах

const date1 = new Date(2021, 3, 4, 5); - получил на вывод 2021-04-04T02:00:00.000Z

Месяцы в датах идут с нуля(0) 0 –январь итд.

02:00:00.000Z - показывает 2 часа, так как наша дата учитывает часовые пояса и выставляет их по Гринвичу. (5-3 = 2). Это формат UTC. Работает для VS консоли. А наш браузер выведет дату, привязанную к нашему локальному времени.

Это выведет консоль 2022-04-15T15:04:55.466Z. А браузер выведет Fri Apr 15 2022 18:02:57 GMT+0300.

Любая дата, которая хранится в JS, хранится в миллисекундах и называется time stamp. Причем количество миллисекунд отчитывает от 1970 года. Поэтому в параметр конструктора мы можем передать миллисекунды. new Date(0); - выдаст нам 1 января 1970 года.

Поэтому любое значение что мы задаем конвертируется в миллисекунды. Это разница между тем что мы задали и 01.01.1970

Чтобы получить дату до 1970 года, нужно использовать отрицательное значение миллисекунд

Методы объекта Date/

Получение компонентов даты

Now.getFullYear() – получить год

Now.getMounth() – получить месяц

Now.getDate() – получить день

Итд getSEconds….

Now.getDay() – получить день недели. Нумерация начинается с воскресенья(0) и суббота (6). Получаем ответ в виде числа. 1 будет понедельник

Все методы выше возвращают значение в соответствии с местным часовым поясом. НО иногда нам может потребоваться часовой пояс, который называется UTC(+0). Поэтому у всех методов выше есть аналоги, которым просто прибавляется UTC. Now.getUTCHours()

Now.getTimeZoneOfset() - получается разница между нашим часовым поясом и UTC. Результат в минутах.

Now.getTime() - возвращает time stamp – количество миллисекунд с 01.01.1970

Установка компонентов даты. Тоже самое только через set

Now.setHours(18, 15); - установка 18 часов 15 минут

Можно также узнать разницу дат так как они в миллисекундах и вычитаются как обычные числа.

(new Date('2022-04-15') - new Date('2022-04-14'))

Date.now() для быстрого получения текущего времени в формате таймстампа.

Семантически он эквивалентен new Date().getTime(), однако метод не создаёт промежуточный объект Date. Так что этот способ работает быстрее и не нагружает сборщик мусора.

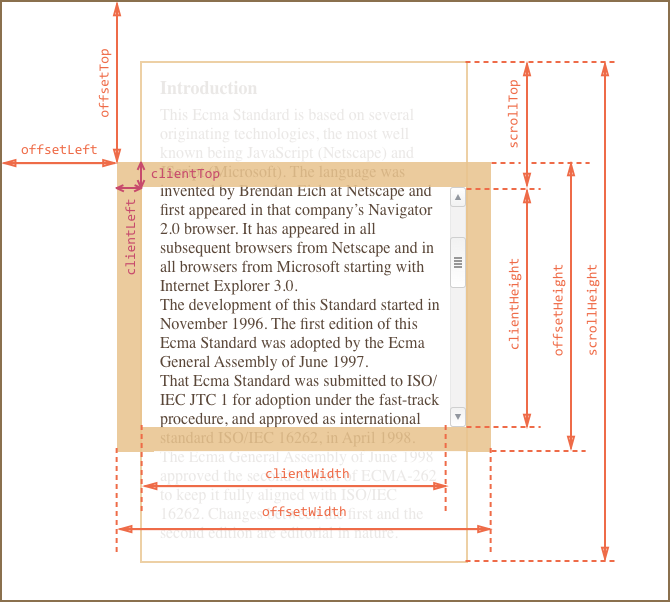
**Параметры документа, окна и работа с ними**

Document – это объект, который содержит всю нашу html структуру.

Window – это объект. Это окно в котором показывается наш document. Тот контент, который мы видим на сайте.

Screen – это объект. Это экран. Весь наш видимый монитор.

Все эти значения измеряются в пикселях. Но в коде JS они пишутся без единиц измерения.



clientWith, clientHeight – ширина или высота элемента включая пединги, но без border и прокрутки справа. Такие параметры нужны в js так как размер в пикселях не всегда можно получить из css так как там может быть все задано в процентах.

offsetHeight, ofsetWidth - ширина или высота элемента включая пединги, бордер и марджин.

scrollHeight – высота всего элемента в целом. Даже если мы его не видим весь на экране.

scrollTop, scrollLeft - Сколько есть пролистанного контента, который не показывается. Эти значения в отличие от предыдущих могут быть модифицированны. Div.scrollTop. Если нам нужно получить это значение для объекта document, то это делается так document.documentElement.scrollTop.

scrollBy(откуда, насколько проскролить)- скролим на определнную величину от отпределнного положения(не обязательно от верха)

scrollTo

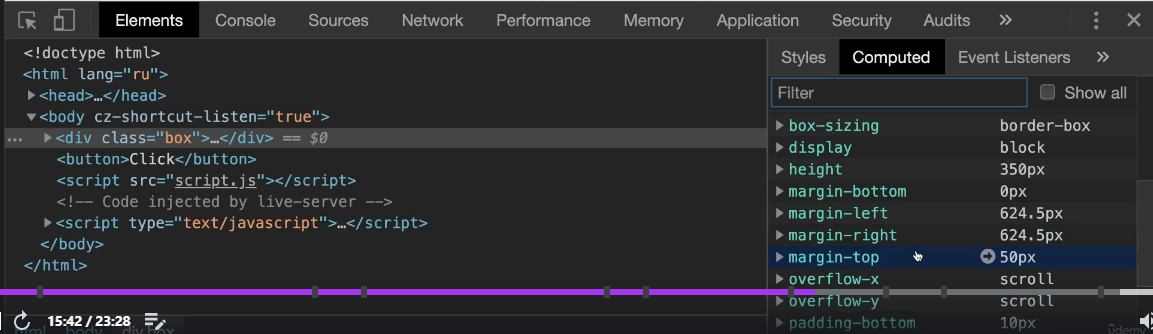
Свойство pageYOffset – показывает сколько пикселей отлистал пользователь сверху. (window. pageYOffset)

В JS значения координат начинается в левом верхнем углу экрана. Div.getBoundingClientRect() – получить координаты элемента

Как получить стили, которые уже применены к элементу при помощи css.

Var abc = window.getComputedStyle(div, второй параметр только для псевдоэлементов);

Стили которые уже были применены к элементу можно посмотреть во вкладке Computed



ComputedStyle – это стили которые идут из CSS и они появятся на странице вначале.

InlineStyle- это стили заданные из js. Они прописываются прямо в тег при помощи JS.

InlineStyle более приоритетные, чем ComputedStyle.

**Создаем модальное окно**

Тригеры – это те элементы, которые вызывают последующее действие. (Элементы на которых висят отбработчики называются тригерами?)

Если у нас разные элементы вызывают какое-то одно действие, то им в тегах часто прописывают какой-то общий дата-атрибут.

keydown – событие срабатывает тогда, когда нажимается кнопка

у объекта-события есть свойство code и мы можем отслеживать его, когда нажимаем на какую-то кнопку клавиатуры. У каждой кнопки есть свой уникальный code.

<https://keycode.info/> - узнать код любой клавиши и не только

В нашей задаче модальное окно это окно которые открывается на весь экран, но мы этого не видим. Мы можем увидить это если наведем на блок <div class="modal">, а вот само окно с информацией находится в блоке <div class="modal\_\_dialog">. Поэтому когда мы кликаем в видимое окно мы попадаем в modal\_\_dialog, а когда в невидимую часть, то попадаем в modal.

П.С. modal\_\_dialog находится внутри блока modal и почему браузер различает в какое окно мы ткули не понятно, так как если рассматривать всплытие событий у нас сработает 2 события если мы кликнем 1 раз на дочерний элемент который находится внутри родительского?

**Модификация модального окна**

Есть событие scroll, которое вешается на глобальный объект window.

Свойство pageYOffset – показывает сколько пикселей отлистал пользователь сверху.

**Функции-конструкторы**

По классификации функция является объектом. По идее в нее можно записать какие-то методы и свойства.

User.prototype.exit = finction(){} – так мы можем добавить прототипу User метод exit и он будет наследорваться у всех потомков. – это для ES5.

Для ES6 можно использовать классы м прописать метод exit прям внутри класса.

Функции-конструкторы являются обычными функциями. Но есть два соглашения:

1. Имя функции-конструктора должно начинаться с большой буквы.
2. Функция-конструктор должна вызываться при помощи оператора "new".

function User(name) {

this.name = name;

this.isAdmin = false;

}

*let user = new User("Вася");*

Когда функция вызывается как new User(...), происходит следующее:

1. Создаётся новый пустой объект, и он присваивается this.
2. Выполняется код функции. Обычно он модифицирует this, добавляет туда новые свойства.
3. Возвращается значение this.

Другими словами, вызов new User(...) делает примерно вот что:

function User(name) {

*// this = {}; (неявно)*

// добавляет свойства к this

this.name = name;

this.isAdmin = false;

*// return this; (неявно)*

}

**Технически любая функция может быть использована как конструктор! То есть, каждая функция может быть вызвана при помощи оператора new.**

## [Возврат значения из конструктора return](https://learn.javascript.ru/constructor-new?ysclid=l6euusihlt596866944" \l "vozvrat-znacheniya-iz-konstruktora-return)

## Обычно конструкторы ничего не возвращают явно. Их задача – записать все необходимое в this, который в итоге станет результатом.

## Но если return всё же есть, то применяется простое правило:

* При вызове return с объектом, будет возвращён объект, а не this.
* При вызове return с примитивным значением, примитивное значение будет отброшено.

Кстати, мы можем не ставить скобки после new, если вызов конструктора идёт без аргументов. Пропуск скобок считается плохой практикой, но синтаксис языка такое позволяет.

let user = new User; // <-- без скобок

// то же, что и

let user = new User();

В this мы можем добавлять не только свойства, но и методы.

function User(name) {

this.name = name;

this.sayHi = function() {

alert( "Меня зовут: " + this.name );

};

}

П.С. не совсем понимаю с этой строчкой. alert( "Меня зовут: " + this.name );

Если мы создали объект this и потом его вернули в другой, то уже другой объект хранит ссылку на данный объект в памяти. Следовательно как мы можем обратиться к this.name? Ведь переменная this уже не ссылается на этот объект и по логике должна быть ошибка. Как вариант туда уже подставляется строка из(this.name) на момент создания этого свойства. Т.е. фактически когда мы обращаемся к этому свойству из другого объекта оно хранит, например:

alert( "Меня зовут: " + “Вася” ); Но это не точно.

<https://learn.javascript.ru/constructor-new?ysclid=l6euusihlt596866944>

**This**

Полная запись метода в объект

user = {

sayHi: function() {

alert("Привет");

}

};

сокращённая запись. В большинстве случаев сокращённый синтаксис предпочтителен.

user = {

*sayHi() { // то же самое, что и "sayHi: function()"*

alert("Привет");

}

};

## **[Ключевое слово «this» в методах](https://learn.javascript.ru/object-methods?ysclid=l6lxwwzqva797175129" \l "klyuchevoe-slovo-this-v-metodah)**

Как правило, методу объекта необходим доступ к информации, которая хранится в объекте, чтобы выполнить с ней какие-либо действия.

**Для доступа к информации внутри объекта метод может использовать ключевое слово this.**

Значение this – это объект «перед точкой», который использовался для вызова метода.

let user = {

name: "Джон",

age: 30,

sayHi() {

*// this - это "текущий объект"*

*alert(this.name);*

}

}

## **[«this» не является фиксированным](https://learn.javascript.ru/object-methods?ysclid=l6lxwwzqva797175129" \l "this-ne-yavlyaetsya-fiksirovannym)**

Значение this вычисляется во время выполнения кода и зависит от контекста.

Например, здесь одна и та же функция назначена двум разным объектам и имеет различное значение «this» при вызовах:

function sayHi() {

alert( this.name );

}

*user.f = sayHi;*

*admin.f = sayHi;*

user.f(); // Джон (this == user)

admin.f(); // Админ (this == admin)

Правило простое: при вызове obj.f() значение this внутри f равно obj.

Вызов без объекта: this == undefined

Мы даже можем вызвать функцию вовсе без использования объекта:

function sayHi() {

alert(this);

}

sayHi(); // undefined

В строгом режиме ("use strict") в таком коде значением this будет являться undefined.

В нестрогом режиме значением this в таком случае будет глобальный объект.

В JavaScript this является «свободным», его значение вычисляется в момент вызова метода и не зависит от того, где этот метод был объявлен, а зависит от того, какой объект вызывает метод (какой объект стоит «перед точкой»).

Udemy

Контекст вызова – это то место откуда вызывается this. Точнее к чему оно привязано. К одному объекту или к другому. Если вообще без объекта, то будет undefined.

Функция может вызываться 4-мя способами и в каждом контекст вызова отличается.

1. Простой вызов функции. This = underfined если используем ("use strict"), или глобальный объект Window, если не используем ("use strict").
2. Контекст у методов объекта – сам объект. Если вызываем obj.method(), то this будет ссылаться на сам объект. Но если вдруг внутри метода method у нас хранится еще один метод и в нем используется this, то такой this не будет ссылаться на объект, так как js будет расценивать это как простой вызов функции(пункт 1) и он будет либо underfined либо Window. Т.е. его контекст будет не объект, а другой метод(в нашем случае method)
3. This в конструкторах и классах – это новый экземпляр объекта.
4. Например, есть функция sayName(surname) и мы можем ее передать в нужный нам контекст вызова (откуда она будет вызываться) 2-мя способами. Функцией call и apply.

sayName.call(user, “Petrov”); Можно также использовать apply - sayName.apply(user,[“Sidorov”]);

sayName – что передаем (наш метод)

user – куда передаем. В какой контекст. В объект user

“Petrov” – аргумент, который попадет в surname функции sayName. Кавычки для call, массив для apply.

call – при помощи чего передаем.При помощи вспомогательного метода call

1. Bind.

Создает **новую** функцию и под нее подвязывает контекст.

Есть функция в которой this не имеет контекст

function func(soname) {

alert(this.firstName + soname);

}

В современном JavaScript у функций есть встроенный метод [bind](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/bind), который позволяет зафиксировать this.

let boundFunc = func.bind(user); - присвоение экзотического объекта в переменную. И установление контеста this=user

boundFunc(“Petrov”) – вызов экзотического объекта.”Petrov” попадает в soname.

func.bind(user) – эта строчка создает особый экзотичский объект, который помещается в переменную boundFunc. И при вызове этот экзотический объект вызывается как функция и прозрачно передает вызов в другую функцию(в func) при этом устанавливая связь this=user.

Другими словами, вызов boundFunc подобен вызову func с фиксированным this.

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

console.log(this);

});

Если мы добавим this в классическую функцию внутри обработччика событий, то при нажатии мы получим в this сам объект на который мы нажали. Тоже самое что и event.target

У стрелочных функций нет своего контекста вызова. Такая функция всегда будет брать контекст вызова у своего родителя.

const obj = {

name: "Igor",

sayName: function(){

let say = () => {

console.log(this.name);

}

say();

}

}

Родитель стрелочной функции, которая помещена в переменную say это другая функция sayName, а контекст функции sayName это объект obj. Т.е. sayName находится в контексте объекта obj. this.name у нас будет Igor. Стрелочная функция взяла контекст своего родителя. Родитель это другая функция, а ее контекст объект. Стрелочная функция находящаяся не в контексте глобального объекта Window называется локальной(например, если находится внутри другой функции или объекта.)

(Объект Window – это глобальный объект в котором хранятся все методы и все переменные. Я так понял это весь наш код?)

Глобальная стрелочная функция в своем контексте всегда будет иметь Window независимо используем мы use strict или нет.

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

console.log(this); // Window

});

Или

Const abc = ()=>{console.log(this);});

Abc(); // Window

https://tproger.ru/translations/javascript-this-keyword/

**Классы (стандарт ES6)**

Если мы используем старые браузеры которые не поддерживают формат ES6, то для них существует трансплиттер, который переводит новый код в старый формат.

Классы – это красивая обертка(синтаксический сахар) фукций-конструкторов. Классы внутри это все теже функции.

Смысл класса это шаблонизация. Всместо того чтобы каждый раз заново описывать какую-то слайдер его свойства, методы итд мы создаем один шаблон, который говрит в общем что будет делать этот компонент.

class Rectangle{

constructor(height, width) {

this.height = height;

this.width = width;

}

calcArea(){

return this.width \* this.height;

}

}

имя класса с большой буквы. Все что внутри будет включаться в этот шаблон. То что будет уметь этот класс.

Constructor(){} – то при помощи чего создается объект класса. Консруктор принимает входящие параметры.

Const obj = new Rectangle(10,12); - при помощи new создаем новый объект this куда все записываем(свойства и методы) и возвращаем наш объект в нашу переменную obj.

Java-учебник

В объектно-ориентированном программировании класс – это расширяемый шаблон кода для создания объектов, который устанавливает в них начальные значения (свойства) и реализацию поведения (методы).

Я так понял, что класс(class) это более продвинутая конструкия функций-конструкторов, которая предоставляет новые возможности.

class User {

constructor(name) {

this.name = name;

}

sayHi() {

alert(this.name);

}

}

// Использование:

let user = new User("Иван");

user.sayHi();

Когда вызывается new User("Иван"):

1. Создаётся новый объект.
2. constructor запускается с заданным аргументом и сохраняет его в this.name.

Методы в классе не разделяются запятой

**Что такое класс?**

В JavaScript класс – это разновидность функции.

*alert(typeof User); // function*

Вот что на самом деле делает конструкция class User {...}:

1. Создаёт функцию с именем User, которая становится результатом объявления класса. Код функции берётся из метода constructor (она будет пустой, если такого метода нет).
2. Сохраняет все методы, такие как sayHi, в User.prototype.

Функции конструкторы отличаются от класса.

1. В отличие от обычных функций, конструктор класса не может быть вызван без new

User(); - так можно вызвать функцию, но не класс

1. Методы класса являются неперечислимыми. И это хорошо, так как если мы проходимся циклом for..in по объекту, то обычно мы не хотим при этом получать методы класса.
2. Классы всегда используют use strict. Весь код внутри класса автоматически находится в строгом режиме.

## **[Class Expression](https://learn.javascript.ru/class?ysclid=l6qqtf4j5s909625783" \l "class-expression)**

Как и функции, классы можно определять внутри другого выражения, передавать, возвращать, присваивать и т.д.

Пример Class Expression (по аналогии с Function Expression):

let User = class {

sayHi() {

alert("Привет");

}

};

new User().sayHi();

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

http://jsraccoon.ru/es6-classes

**Наследование**

class ColoredRectangle extends Rectangle{

constructor(height,width,text,color){

super(height,width);

this.text = text;

this.color = color;

}

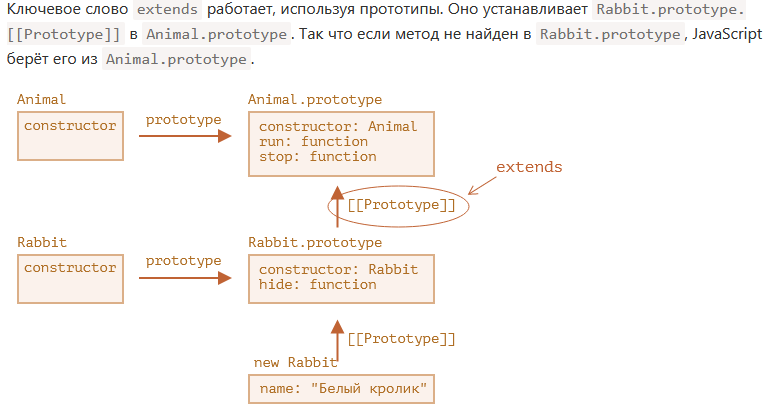
showMyProps(){ return `Text: ${this.text}, Color: ${this.color}`}

}

Extends – показывет что класс наследуется от другого. Т.е объект такого класса будет иметь все свойства и методы родителя, а также свои.

super(параметры что и в родительском конструкторе); - всегда должен быть на первом месте в конструкторе. метод который вызывает супер конструктор родителя. Вызывает тоже самое что и было у родителя в конструкторе. Все присваивания.

Java-учебник



## [Переопределение конструктора](https://learn.javascript.ru/class-inheritance" \l "pereopredelenie-konstruktora)

Если у наследуемого класса не прописан конструктор.Если класс расширяет другой класс и не имеет конструктора, то автоматически создаётся такой «пустой» конструктор:

class Rabbit extends Animal {

// генерируется для классов-потомков, у которых нет своего конструктора

*constructor(...args) {*

*super(...args);*

*}*

}

let rabbit = new Rabbit("Белый кролик"); - при создании объектов используем конструктор родителя

Если мы в Rabit класс прописываем свой конструктор, то при создании конструктора вначале должен быть метод super()- он вызывает родительский конструктор. Дело в том что родительский конструктор создает объект this, дочерние классы этого не делают. Поэтому если в дочернем классе прописать this.name вместо super(), то будет ошибка, так как this никем не создан.

Почитать если что. Тут про наследование, и переопределение, и про другое.

<https://learn.javascript.ru/class-inheritance>

**Rest оператор и параметры по умолчанию**

Если spread оператор брал сущность и раскладывал ее на отдельные элементы, то Rest отдельные элементы объединяет в массив.

Если мы не знаем точно сколько значений придет в параметры функции, то это дело контролирует rest оператор.

Rest оператор записывается всегда последним в функции.

function sum (a, b, ...rest){

}

...rest – рест оператор собирает все оставшиеся значения которые будут переданы в качестве параметров и формирует из этого массив.

Даже если мы ничего не передаем в rest оператор, то онформирует пустой массив. То есть мы можемобратиться к rest всегда.

Параметры по умолчанию

function sum (a, b = 2){

}

Если вместо b не передано явно какое-то значение, то по умолчанию возмется 2.

**Локальные сервера**

Сервер – это программа, котораяпозволяет вамзапускать вебсайты и выполнять более сложные манипуляции с сайтом. (Сервер – это backend часть)

Когда создаемновое приложение, тохотимпротестировать весь функционал, который есть в нашем продукте и для этого нам нужен локальный сервер (иммитация удаленного сервера).

Live server - нажимаем кнопку GoLive(если установлен плагин) и у нас стартует небольшой локальный сервер. Поддерживает только get запрос.

Самое главное что нас интересует это какие запросы умеют обрабатывать эти сервера.

Htttp – запросы (самые популярные get и post)

Get запрос – когда в поисковике вводим какую-то информацию, то мы хотим ее получить с сервера(например Гугла) get – значит получить. После нажатия интер мы получаем эту информацию. Гет запрос – это запрос на получение данных у какого-то сервера.

Post запрос – постит какие-то данные на сервер. У нас есть контактная форма. Мы вводим данные и нажимаем отправить.

**JSON формат передачи данных, глубокое клонирование объектов**

JavaScriptObjectNotation - является тексотовым форматом обмена данных. Используется для передачи и для хранения данных.

JSON – это набор пар ключ-значение. Значения должны быть обязательно в двойных кавычках(“”). В качестве значений могут быть объекты, массивы, числа, строки, логическое значение или null.

Допустим у нас есть объект и мы должны отправить на сервер. Напрямую мы отправить объект на сервер не можем, так как сервер и протоколы передачи данных нас не поймут. Нужно передать его в формат, который мы можем транспортировать.

Все современные браузеры имеют встроенные объекты json для работы с такими данными. И в этих встроенных объектах есть как свойства так и методы. И такой метод stringify превращает объект js в нужный нам формат

JSON.stringify(person);

JSON – это идет обращение к объекту, который встроен в браузер

stringify(person) – метод который преобразует js объект person в формат json

На выходе получается json объект и все его сущности(ключ и значение) записаны в двойных кавычках {“name”:”Alex”,”tel”:”334345”}. Такой формат данных можно отправить на сервер.

Когда с сервера приходит json и мы должны превратить его js объект:

JSON.parse(json object) – метод по переводу из jsona в js.

До появления json у нас использовался язык xml

{"name":"Alex","tel":"334345"} – это json

То же самое в xml. Структура весит больше.

<?xml version="1.0" encoding="UTF-8" ?>

<root>

<name>Alex</name>

<tel>334345</tel>

</root>

**Полное клонирование объектов.**

Const clone = JSON.parse(JSON.stringify(person));

Перевод в json формат, а потом обратно позволяет создать полный клон объекта на всех уровнях вложенности.

**Асинхронное выполнение задач на странице и общение с сервером (AJAX)**

Когда мы что-то выбираем в интерент магазине мы можем использовать фильтры (по моделям, например) и как только мы выбрали галочку то контент на странице меняется. Но мы не перезагружаем всю страницу, а меняем лишь часть контента на странице.

Преимущества

1. Это удобно и красиво. Происходит интерактивная обработка(в режиме реального времени) абсолютно любых событий.
2. Происходит ускорение реакции интерфейса. Мы не перезагружаем каждый раз страницу, а меняем лишь часть контента на странице.
3. Уменьшаем нагрузку на сервер. И экономим трафик пользователя, так как обновляем не всю страницу целиком.

Технология AJAX (asinhronius javascripnt and xml)

Чтобы наша страница(наш фронтенд) умел общаться с сервером нам нужны http запросы. Мы можем использовать get/post итд и чтобы все это происходило асинхронно (без перезагрузки страницы) нам нужны технология ajax.

1-й вариант ajax (реализуется при помощи встроенного в браузер объекта xml http request) – не актуален на данный момент.

События cnange и input.

Событие change возникает когда наш инпут уходит из фокуса.(Было подсвечено какое-то текстовое поле(инпут) и мы в него что-то написали. Как только мы убрали с него фокус(переключились на другой инпут или просто щелкнули по пустой странице), то сработало событие change. Срабатывает как только мы убираем фокус с нашего элемента. Или когда меняется value внутри инпута. Двояко он объяснил.)

Событие input происходит каждый раз когда что-то вводится или удаляется внутри инпута. Написали 1 имвол сработало событие, дописали второй символ опять сработало, удалили первый символ опять стработало. Срабатывает на каждоое измененение внутри инпута.

Чтобы сделать запрос на сервер нужно использовать встроенный объекта в браузер xml http request. Создается он const request = new XMLHttpRequest(); Такой объект имеет свои методы и свойства.

request.open(method, url, async, login, pass); - метод собирает настройки, которые помогут сделать запрос

method – тот метод что используется для запроса get/post. ’GET’’POST’ – всегда записываются в верхнем регистре

url – путь к нашему серверу.

Запросы мы отсылаем из html файла, так как наша страница открыта в браузере. Формируем путь относительно html страницы. Js/current.json – такой путь прописываем до файла, так как наш html файл лежит в папке js.

async – отвечает за асинхронность. По умолчанию true. (синхронный код идет попорядку. Выполняется одно, потом другое. Асинхронный код выполняется параллельно с другим кодом. Асинхронный код не блокирует остальной.) Ajax запросы по умолчанию являются асинхронным кодом. Мы посылаем запрос на сервер. Мы не знаем сколько времени сервер нам будет отвечать. В это время продолжает выполняется другой код js, который после этого запроса. И когда сервер ответит, то наш асинхронный запрос закончит свою работу. Если async = false, то пока вам сервер не ответит, другой код не будет выполняться. После ответа с сервера завершается метод запроса и потом идет другой код.

login, pass – логин и пароль. Некоторые запросы мы можем делать только с ними

request.setRequestHeader(‘Content-type’,’application/json; charset- utf-8’); - метод указывает,что именно отправляем. Нужно для того чтобы транспортные протоколы знали что передают, а сервер точно знал, что он принимает.

Content-type – тип контента

’application/json; charset- utf-8’ - указываем тип контента для json файла. А также можем указать кодировку.

request.send(body); - метод отправки запроса на сервер.

Body – те данные которые уходят на сервер. Используем только в методах POST

Свойства объекта new XMLHttpRequest();

Status – показывает статус нашего запроса (404,0,200 итд)

statusText - текстовое описание ответа от сервера

response – ответ от сервера. Тут лежит ответ, который нам задал backend разработчик, какую инфомаию мы должны использовать на клиете

readyState – содержит текущее состояние нашего запроса

События связанные с запросом

request.addEventListener('readystatechange',()=>{}) – событие отслеживает статус нашего запроса в текущий момент. Следит засвойством readyState. Как только значение это свойства меняется у нас срабатывает это событие.

Если сервер ломается или что-то пошло не так, то это отображается при помощи

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

console.log(request.response);

} else{

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

}

request.addEventListener('load',()=>{}) – событие срабатывает 1 раз когда наш запрос полностью загрузился и мы получили результат. Срабатывает даже когда status != 200.

form.addEventListener('submit ',()=>{})– событие срабатывает каждый раз, когда мы пытаемся отправить какую-то форму. П.С. если в верстке кнопка задана тегом button, то у нее автоматически стоит type = submit.

Также нужно незабыть выполнить внутри обработчика e.preventDefault(); // отменяет стандартное поведение браузера. В данном случае отменяет перезагрузку страницы при отправке формы.

New FormData(form) – специальный объект который позволяет с определнной формы быстро собрать(сформировать) все данные в формате ключ-значение, которые заполнил пользователь. form – откуда собираем данные.

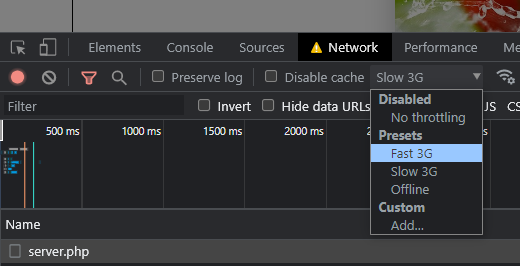
Когда создаем верстку (input, options, textarea итд) и подразумеваем, что эти данные пойдут на сервер, то мы всегда у такой верстки должны указывать атрибут name=”что-то”. Иначе FormData не сможет найти этот инпут и взять из него value.

Во вкладке браузера Network перейти в Request Payload – там отображаются данные которые мы отправляли на сервер.

Когда мы используем связку из объекта new XMLHttpRequest(); и объекта new FormData(), то нам заголовок устанавливать не нужно(он устанавливается автоматически). А если мы используем заголовок, например - request.setRequestHeader('Content-type', 'multipart/form-data') , то **сервер данные не получит.**

Проверить что сервер получил данные можно посмотрев массив или в гугл ром посморев вкладку Headers – Form Data

В гугл хром браузере также есть удобная вещь для тестирования. Можно выставить медленную работу интернета и тогда видеть все процессы по выполнению кода более подробно.



Если наш сервер должен прнимать данные не в обычном формате, а в json.

Получить значение атрибута

<div id="elem" about="Elephant"></div>

alert( elem.getAttribute('About') ); // (1) 'Elephant', атрибут получен

Если данный атрибут не существует у указанного элемента, то возвращаемое значение будет соответствовать значению null.

**Promise (ES6)**

Promise технология позволяет работать с различными асинхронными операциями. (например в таймаутах или запросах на сервер). Promise – это обещание.

Часто в коде мы хотим выполнить последовательность каких-то действий. Если произошло что-то, то мы обещаем, что у нас выполнится следующее действие.

Если у нас setTimeout содержит в себе другие setTimeout как матрешка, то такой код очень сильно разрастается и его неудобно читать и анализировать. В таком коде колбеки позволяют соблюдать определнный порядок этих действий. Чтобы облегчить код у нас существуют Promise.

Let req = new Promise(function(resolve, reject){}); - создаем обещание внутри переменной req. Внутри колбек функция. Содержит2 аргумента resolve, reject.

Resolve – что-то выполнилось правильно

Reject – что-то пошло не так. Обещание не выполнилось.

req.then(); - метод, который выполняется на промисе в случае положительного исхода - это функция resolve(). Если я правльно понял, то req.then() – это и есть resolve. Внутри req.then() мы описываем набор действий, который сработает в коде, когда вызовется функция resolve(). Т.е. дебаггер переместиться внутрь функции req.then();

req.then(product); - может содержать в виде параметров данные(product),которые например приходят с сервера. И потом с ними работает.

1. Создаем обещание внутри переменной req.
2. Когда мы создаем обещание мы предполагаем, что оно может завершиться как положительно так и отрицательно. У нас есть определнный промежуток времени до того как мы получим результат.
3. Внутри промиса есть 2 аргумента resolve, reject. Это аргументы вместо которых будут подставляться функции. Если все хорошо(сервер нам ответил) то будет выполняться функция resolve. Если что-то не так то вызовем reject.

У Promise есть огромное преимущество перед callback. Мы можем возвращать Promise и then по цепочке. Когда одна асинхронная операция выполнится , то мы выполним следующую, потом опять следующую итд.

req.then((product) => {

return new Promise((resolve, reject)=>{}); - можем возвращать промис

// return data; - а можем возвращать просто данные и по цепочке с ними работать

}).then((data)=>{}) – обработка промиса по цепочке

Тут наш промис req возвращает другой промис и сразу у него(у этого нового промиса) мы можем вызвать метод then(). Без объявления имени промиса. Это называется **цепочка**. Мы можем создавать настолько длинную цепочку, насколько нужно. **Т.е. асинхронный код будет четко следовать друг за другом.**

req.catch(()=>{}); - если наше обещание не выполнилось(ошибка на сервере) сработает этот код. Тут описывается функция reject(); Если у нас ошибка, то мы пропускаем все then и перемещаемся в catch.

Блок req.finally(()=>{}); блок располагается в самом конце. Он выполняется при любом исходе(выполнилось обещание или нет)

Promise.all([]) – служит для того чтобы мы убедились, что все наши промисы уже выполнились. Если мы например делаем запросы к разным серверам, а ответы с них хотим использовать одновременно. И вот чтобы эти промисы ждали друг друга сузествует такой метод. Потом мы их все можем обработать через then

Promise – как глобальный объект

all([]) – метод all, который принимает массив с промисами

Promise.race([]).then() – метод race выполняется сразу как какой-то из промисов уже отработал.

/////////////////

Из разных источников

[Promise](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise) (промис) - это объект, представляющий результат успешного или неудачного завершения асинхронной операции.

В сущности, промис - это возвращаемый объект, в который вы помещаете колбек.

Let prom = new Promise((resolve, reject) => {if()resolve(data) else reject (error)});

Этот колбек принимает 2 аргумента resolve, reject. Эти аргументы это тоже методы. И вызываются они в зависимости от того успешно ли завершился запрос на сервер. Если успешно, то по завершении запроса с сервера выполняется метод resolve, если запрос на сервер завершился не успешно, то по звершени запроса метод reject. Делается это для того чтобы показать что промис завершился успешно или нет.

Если запрос завершился успешно вызовем resolve(data). В качестве аргумента data передается ответ с сервера(response). Если вы зывается reject(error), то в качестве параметра можем передать ошибку с сервера.

Внутри промиса описывается лишь условие когда будет вызван resolve, reject. Но не само тело этих методов. Само тело прописывется отдельно. Для resolve через метод then, для reject через метод catch.

prom.then((res) => {console.log(res)} - тело метода resolve. Т.е. наша data попадет в res

.catch((err) => {console.log(err)}) - тело метода reject. Т.е. наш error попадет в err

Цепочка методов

prom.then((res) => {

console.log(”Рузультат:”, res)} // выведет Рузультат: + что-то вместо res

}).then((res1)=>{

console.log(”Рузультат:”, res)} // выведет Рузультат: + undefined

})

Дело в том что первый then получает свой res который передается как data внутри промиса prom(смотри описание выше). Но дальше метод then после выполнения по умолчанию передает совершенно новый промис.

Тут наш промис prom возвращает другой промис и сразу у него(у этого нового промиса) мы можем вызвать метод then(). Без объявления имени промиса. Это называется **цепочка**. Мы можем создавать настолько длинную цепочку, насколько нужно. **Т.е. асинхронный код будет четко следовать друг за другом.**

И уже в этом новом промисе параметр res1 ничего не получает и поэтому его значение будет undefined. Чтобы этого избежать мы можем передать какие-то данные через return.

Например,

prom.then((res) => {

// return “123”; - а можем возвращать просто данные и по цепочке с ними работать

}).then((res1)=>{

console.log(”Рузультат:”, res)} // выведет Рузультат: + 123

})

**Метод Then – предполагает что любой возвращенный нами ответ будет автоматически обернут в Promise**

Т.е. мы сделали запрос на сервер, получили с него какой-то ответ, сделали манипуляции с этим ответом и передали его дальше по цепочке. Мы можем взять какие-то данные из ответа и использовать эти данные для запроса на другой сервер, получить данные с него и передать дальше.

П.С.если вначале указываем catch, а потом then методы и у нас срабатывает блок catch, то все then методы после него выполнятся.

Если мы сначала указали все then методы, а потом catch и у нас срабатывает блок catch, то все then методы пропускаются и выполняется catch, а потом все что после него.

Если у нас есть несколько блоков catch и мы хотим попасть во все этиблоки то мы должны использовать в них вместо return выражение throw ‘данные что передаем дальше’

////////////////////////

Promise (промисы, обещания) – это специальный объект и набор методов в JavaScript для удобного написания асинхронного кода.

В отличие от обратных вызовов промисы позволяют решать асинхронные задачи значительно проще без создания большого количества вложенностей одних функций в другие.

Просмис – это некое обещание. Мы обещаем в будущем что-то выполнить или нет в зависимости от условий. Я обещаю дать сыну 100 $ если он в будущем получит 5 за экзамен или ничего не дам, если получит не 5.

В данный момент вы находитесь в состоянии ожидания, т.к. не знаете какую отметку вы получите, а следовательно, не знаете получите ли вы бонус в размере 100$.

Но как только вы получите отметку по экзамену, обещание завершится. Далее в зависимости от того успешно оно завершилось или нет будет зависеть получите ли вы 100$ или нет.

**Как выполняется промис**

const passexam = true;

// промис

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

setTimeout(() => {

passexam ? resolve('Папа подарил 100$.') : reject('Папа не подарил 100$.');

}, 10000);

});

console.log('222');

result

.then(value => {

console.log(result);

console.log(value);

})

.catch(value => {

console.log(result);

console.error(value);

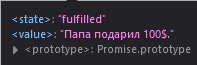
});

console.log('333');

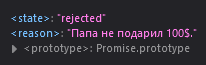
Сначала попадаем в строку const result = new Promise. Тут инициализируется новый промис, значение = undefined. Далее запускаем строку setTimeout(() выполнения асинхронного кода, в этот момент значение = undefined. Далее мы выходим из промиса и продолжаем деалать наш синхронный код переходя к строчке console.log('222'); Как только мы вышли из промиса, то состояние у него меняется на state: pending – ожидание. Т.е. наш промис ждет выполнения асинхронного кода, ждет ответа. В это время продолжается выполнятся синхронный код и мы переходим к строке console.log('333'); (и далее идем по коду, если бы были еще строки).

Как только ответ получен (например setTimeout выполнился или пришел ответ с сервера) выполнится одна из функций в зависимости от ответа resole() или reject(). Как только выполнилась одна из этих функций это поменяет состояние нашего промиса.

Выполнилось resole(),



Если выполнилось reject()



Теперь наш промис завершился и содержит какой-то ответ(положительный или отрицательный) Далее мы переходим к строке с методами then catch и в зависимости от этого ответа будет вызываться метод then или catch. В value хранится то, что мы передали в качестве параметра в resole(Папа подарил 100$) или reject(Папа не подарил 100$.).

Краткая версия. Пока промис ожидает ответа он пендинг. Как только ответ пришел он становится fulfield или rejected. Далее в зависимости от этих ответов выбирается метод then или catch.

<https://itchief.ru/javascript/promise?ysclid=l7zwbpjwlb83489237>

//////////////////////////////////////////////////

Михаил Непомнящий

В ручную промисы создавать приходится очень редко.

После returne new Promise в памяти создается некий специальный объект

{

Value:undefined

onFulfielf:[] – пустой массив изначчально

onRejected:[] – пустой массив изначально

}

Далее then и catch заполняют массивы в полях onFulfielf onRejected. Например в них лежит console.log и console.error.

Далее промис по своей логике(она может быть скрыта) вызывается либо resolve(data) либо rejected(err).

Если вызывается resolve(data), то data подставляется в наш объект промис. Теперь он выглядит так.

{

Value: data

onFulfielf:[console.log] – заполнено на этапе then

onRejected: [console.error] – заполнено на этапе catch

}

Как только значение data попадает в объект начинает отрабатывать onFulfielf.

Если у нас цепочка методов, то при вызове следующего then новое значение помещается в value (то которые было передано через returne из предыдущего then) и снова отрабатывает onFulfielf(но уже следующая функция которая в него была запсана, так как это массив)

https://www.youtube.com/watch?v=4X7OjIjjaQE

**Fetch API**

API – Application Programming Intarface. Это решение, которое предоставляет готовые методы и свойства, которые мы можем использовать.

Fetch API – уже встроена в браузер. Технология, которая умеет общаться с сервером и построена на промисах. Эта технология предоставляет, готовые методы.

fetch(url)- метод для запросов с сервера.

url – куда посылаем запрос

<https://jsonplaceholder.typicode.com/> - это небольшая БД в формате json, которая лежит в интернете и к который мы можем обращаться для тестирования.

Вот запрос get запрос

fetch('https://jsonplaceholder.typicode.com/todos/1')

.then(response => response.json())

.then(json => console.log(json))

Наш fetch использует промисы. Из этой конструкции fetch('https://jsonplaceholder.typicode.com/todos/1') возвращается промис. И мы сразу можем его обработать при помощи then()

В response помещается ответ с сервера в формате json.

response.json() – а вот этот метод превратит данные из json обычный js объект. Также эта команда возвращает нам промис и мы дальше можем построить цепочку.

Если добавить дополнительные параметры, то метод превращается в POST запрос.

fetch(url, {объект с настройками})- метод для запросов на сервер.

fetch('https://jsonplaceholder.typicode.com/posts',{

method: "POST", // метод запроса на сервер

body: JSON.stringify({name:'Alexey'}), // то что отправляем на сервер, если нужно

headers: {

'Content-type': 'application/json'

} // Заголовки, которые определяют какой контент мы отправляем. так как заголовкИ, то мы используем объект.

})

.then(response => response.json())

.then(json => console.log(json))

\*\*\*\*П.С. От себя. Скореее всего.

Когда передаем промисы по цепочке, то каждый раз вызываем новый промис внутреннюю логику которого не видим.

fetch('https://jsonplaceholder.typicode.com/todos/1') – вернет промис внутри которого в resolve(response) подается респонс, но мы этого невидим и незнаем наверняка. Потом этот response передается в then где с ним что-то делаеют.

.then(response => response.json()) – этот then снова возвращает промис внутри которого resolve(обект в js формате) помещается результат response.json() те.е объект в js формате котоорый попадет в следующий then. Но опять же мы этого не видим и не знаем наверняка.

then(json => console.log(json)) – выполняем код внутри, а потом тут я так понимаю тоже возвращается промис, но then мы у него не вызываем и поэтому не используем последний промис.

\*\*\*\*

**Если внутри fetch промис попадает на ошибку с сервера котрая связана с http протоколом (например ошибка 404, 500 итд), то fetch не вернет reject, он вернет resolve. Это для него не считается ошибкой. Самое главное для fetch что он вообще смог сделать этот запрос(выполнился код). Rejec будет возникать при сбое сети(отключен интернет итд) или что-то что помешало запросу выполнится.**

**Методы перебора массивов**

forEach() – не возвращает новый массив. Берет какой-то массив и просто перебирает.

**filter**

Метод фильтрует элементы внутри массива. **Только фильтрует**. Мы выбираем хотим ли мы получить данный элемент целиком или нет. Метод возвращает новый массив.

filter(function callbackFn(element, index, array) { ... }, thisArg)

element - Текущий обрабатываемый элемент в массиве

index - Индекс текущего обрабатываемого элемента в массиве.

Array - брабатываемый массив, на котором был вызван метод filter()

thisArg - Значение, используемое в качестве this при вызове колбэк-функции callbackFn

П.С. если у нас есть массив массивов и мы его перебираем мы не можем взять какую-то часть от массива, а только весь массив целиком, если он нам нравится. Если хотим взять часть, то это уже метод map

const arrName = ['aaa','ffffff','wewewe'];

const filteredName = arrName.filter((name) => {return name.length < 5;});

колбек – перебирает элементы и возвращает только те элементы которые подходят под условие.

Первый агрумент(name) –это каждый элемент массива в отдельности.

**Map**

*arr*.map(function *callback*( *currentValue*[, *index*[, *array*]])

Метод map позволяет взять исходный массив и изменить каждый элемент внутри него. Мы можем взять элемент и вернуть его какую-то часть или сделать еще что-то. Вернет новый массив.

const arrName = ['aAa','ffffFFFf','weWWewe'];

const filteredName = arrName.map(item => item.toLocaleLowerCase());

**some**

Перебирает массив и если хотя бы один элемент подходит под условие, то в таком случае он нам вернет true, если нет, то false.

const arrName = ['aAa',123,'weWWewe'];

console.log(arrName.some(item => typeof(item) === 'number')); // true

**every**

Перебирает массив и если все элементы подходят под условие, то в таком случае он нам вернет true, если нет, то false.

const arrName = ['aAa',123,'weWWewe'];

console.log(arrName.every(item => typeof(item) === 'number')); // false

**Reduce**

Возвращает новый реультат. Метод собирает массив в одно единое целое.

Например с цифрами(также мы можем соединять строки и другие элементы массива)

const arrName = [1, 16, 32];

console.log(arrName.reduce((sum, item) => sum + item));

Проходим по массиву и суммируем каждый элемент в sum

arrName.reduce((accumulator, currentValue) => accumulator + currentValue, начальное знач.для sum));

При первом вызове функции, параметры accumulator и currentValue могут принимать одно из двух значений. Если при вызове reduce() передан аргумент initialValue, то значение accumulator будет равным значению initialValue, а значение currentValue будет равным первому значению в массиве. Если аргумент initialValue не задан, то значение accumulator будет равным первому значению в массиве, а значение currentValue будет равным второму значению в массиве.

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global\_Objects/Array/Reduce

**entries**

**Object.entries()** метод возвращает массив собственных перечисляемых свойств указанного объекта в формате [key, value]

Object.entries(наш объект) – в данном слуычае создает массив из ключа и значения объекта и помещает в отдельный массив и так по всем свойствам. Возвращает массив.

Позволяет взять объект и превратить его в матрицу. Т.Е. в массив массивов

**npm и проект. JSON-server**

npm пакеты – это кусочки какого-то кода(компиляторы css, task runner,сборщик проектов итд), которые лежат на отдельных серверах, которые мы можем устанавливатьсебе в проект.

Но прежде чем это сделать мы должны нашей системе сказать что наш проект(который открыт в редакторе кода) будет содержать npm пакеты. Он превращается в npm проект. Это нужно чтобы наша система четко знала что в этом проекте есть, какие пакеты он использует, какая версия и кто автор.

Прописыываем в терминале. Убеждаемся что адрес проекта верный.

1. Поэтому сначала создаем npm проект – npm init.

Указываем имя нашего проекта package name: (имя проекта)

Далее version,description, entry point, test command,git repository,keyword, author, lisence – пока пропускаем. Создается объект с этими настройками, которые будут применены кнашему проекту. Is this Ok ? вводим “y”

У нас в проекте появился файл, который называется package.json с нашими настройками. Он будет содержать информацию о нашем проекте и о тех npm пакетах, которые мы поместим в наш проект.

1. Установка npm пакета. (установка json server)

npm i json-server --save-dev команда для установки

i – install

json-server – название пакета

дополнительные параметры при установке. Пакеты можно устанавлвать локально и глобально. Если пакет установим глобально, то он будет работать для всех проектов. Это полезно для проверяющих утилит. Но чащепакеты устанавливаются локально, для конкретного проекта. Любой другой пользователь который будет пользоваться нашим проектом будет знать что ему нужно доустановить, если вдргу потребуется, не взирая на то что было установлено глобально у предыдущего пользователя.

Для локальной установки ничего не указываем “”

Для глобальной установки указываем флаг “-g”

Далее указываем используется ли этот пакет при разработке или при работе с проектом.

--save-dev - такой ключ говрит что пакет используется только при разработке

--save - такой ключ говрит что пакет используется при работе проекта Пакеты использующиеся приработе (разные библиотеки, jquery, react итд)

После установки пакета в файле package.json у нас появилос свойство с нашим npm пакетом devDependencies : json-server 0.16

А если бы мы устанавливали пакет не для разработки а для работы(--save), то json-server 0.16 просто бы появился в package.json без свойства devDependencies.

Также появилась папка node\_modules – там все npm пакеты, которые мы устанавливаем в наш проект. Там находится json-server, а также различные зависимости, которые он подтягивает к себе. **Папку node\_modules нельзя трогать и что-то там изменять, копировать или сохранять!** Это npm пакеты, которые работают непосредственно с node-js.

Папку node\_modules не нужно пушить! Она может весить 300-500Мб. Прописываем ее в gitignore. П.С. эту папку мы неудаляем вручную если захотим! Для этого есть специальная команда.

Когда другой пользователь скачает проект с удаленного репозитория, там не будет папки node\_modules. Но ему она нужна, так как он должен пользоваться всеми зависимостями. Для этого есть команда npm i. Эта команда возьмет файл package.json и установит пользователю все зависимости, которые прописаны в этом файле. Главное чтобы был файл package.json.

Также появился файл package-lock.json – тут записаны минизависимости длябольших пакетов, которые мы устанавливаем. Этот файл тоже не трогаем!

Json-server – это простой сервер по работе с json файлами, когда мы их можем использовать как маленькую БД. Мы можем делать post запросы и организовывать их в правильные json файлы.

**Хорашая штука. Связь JQuery и js**

https://only-to-top.ru/blog/coding/2019-09-24-jquery-to-js.html

1. Как подключать js
2. Объявление переменных
3. Классификация типов данных
4. Простое общение с пользователем
5. Интерполяция ES6
6. Операторы в JS
7. Условия
8. Циклы
9. Функции, стрелочные функции.
10. Методы и свойства строк и чисел
11. Callback функции.
12. Объекты. Деструктуризация объектов.
13. Массивы и псевдомассивы.
14. Передача по ссылке или по значению, Spread оператор
15. Основы ООП, прототипно-ориентированное
16. Консоль разработчика, Breakpoints.
17. Динамическая типизация
18. Получение элементов со страницы
19. Действия с элементами на странице
20. События и их обработчики.
21. Навигация по DOM элементам
22. Работа с неоптимизированными страницами
23. События на мобильных устройствах