Методы и свойства строк и чисел

Методы – это вспомогательные функции, а свойства – вспомогательные значения. Когда мы используем свойство, то его не вызываем, а просто записываем через точку. Методы уже, напротив, вызываются и прописываются с ().

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

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

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

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String> - Документация по строкам

**Свойство и методы строк:**

* Length – свойство, определяющее кол-во элементов внутри строки/массива
* toUpperCase() – метод строки, который ставит все символы в вехний регистр
* toLowerCase() – метод, противоположный методу toUpperCase()
* indexOf(“word”) – метод для поиска подстроки, позволяет определить есть ли в строке искомое значение и его место в строке по индексу
* slice(6, 11) – метод, возвращающий элементы из строки и принимающий в себя два значения: start – индекс начала строки, с которой необходимо возвращать что-то, end – индекс конца строки (указываем на индекс больше, так как он не будет включаться). Если указать только start, то наша строка будет вырезаться от этого элемента до конца. Поддерживает и отрицательные аргументы индекса, тогда начало будет с конца строки.
* substring() – аналог метода slice(), также принимающий в себя значения start и end. Отличается тем, что start может быть по индексу больше, чем end, также метод не поддерживает отрицательные значения
* substr() – аналог двух предыдущих метод. Отличается вторым аргументов end, принимающий в себя не конец строки, а длину, т.е. сколько символов надо вырезать
* trim() – метод строк, который удаляет пробелы с начала и конца строки. Если строка состоит только из пробелов, то метод позволит вернуть пустую строку

**Свойства и методы чисел:**

Для работы с числа в JS есть отдельная мини-библиотека, которая существует прямо внутри браузера и внутри JS – Math

* Math.round(num) – округляет значение до ближайшего целого
* parseInt() – метод переводит число в другую систему исчисления. Часто используется не по назначению, чтобы перевести строки в целые числа. Вызывается обычно на числах, но можно также использовать как отдельную глобальную функцию
* parseFloat() – метод используется для того, чтобы взять строку/число и вернуть нам его в десятичном исчислении. Возвращает числа с плавающей точкой

Callback – функция

Функция, которая должна быть выполнена после того, как другая функция завершила свое выполнение. Позволяет избежать задержек в функциях и их правильную отработку при вызове.

В другую функцию в качестве аргумента передаётся другая функция – это главная особенность использования callback-функций. Это нужно для того, чтобы функции выполнялись последовательно, строго одна за другой

**Примеры использования:**

Изображение выглядит как текст

Автоматически созданное описание

В данном примере в качестве callback-функции была передана анонимная функция, которая, после завершения выполнения, исчезнет.

Изображение выглядит как текст

Автоматически созданное описание

Во втором примере была передана уже осознанная функция. ВАЖНО! Передаем функцию без (), так как она будет вызываться непосредственно в функции LearnJS в нужный момент!!!

!!!Иногда в callback-функциях нам приходится отходить от правил и передавать аргумент в ней, тем самым вызывая её, и чтобы обойти это, мы можем прописать стрелочную функцию!!!

Изображение выглядит как текст, устройство, датчик, панель управления

Автоматически созданное описание

Объекты, деструктуризация объектов(ES6)

Объекты – это структуры, которые могут сохранять в себе любые типы данных в формате {ключ:значение}. Они могут быть вложенны: объекты в объекты, массивы в объекты и т.д. Создать объект мы можем через {} или:

const obj = new Object();

Обычно данным способом не пользуются, лучше задавать объекты через {}.

Чтобы **достучаться до какого-то свойства** объекта, к нему обращаются или через точку или через []:

Изображение выглядит как текст

Автоматически созданное описание

Например, чтобы **получить свойство** name, мы обратимся к нему как console.log(options.name) или console.log(options[name]). Первый вариант более предпочтительный.

**Для удаления** какого-то из свойств объекта используется delete. Например, delete options.name;

**Для перебора всех свойств** объекта используется конструкция for in



Каждое свойство необходимо как-то назвать, поэтому внутри конструкции создаем переменную key, которая и будет означать каждое свойство объекта. Далее прописывает in и название объекта (options), в котором будет осуществляться перебор свойств. А в {} уже прописываем действие, которое будет совершаться при переборе, в данном случае через конкатенацию выводим строку с парой { ключ: значение }. И получаем результат:

Изображение выглядит как текст

Автоматически созданное описание

Последняя строчка говорит нам о том, что в последнем ключе его значением является объект, и т.к. JS не может такой объект превратить в строку, то выдаёт такой результат. По факту это строковое представление объекта. Мы можем избежать такого используя рекурсию или, если зададим ещё один перебор внутри другого с условием:

Изображение выглядит как текст

Автоматически созданное описание

В таком случае результат будет иным:

Изображение выглядит как текст

Автоматически созданное описание

Есть ещё один способ перебора for of НО! Он не работает с объектами! Данный способ работает только с массивоподобными сущностями (массив, строка, псевдомассив, map и set)

**Упорядочивание свойств объекта:**

Если мы будем в цикле перебирать все свойства объекта, получим ли мы их в том же порядке, в котором мы их добавляли? => Свойства упорядочены особым образом: свойства с целочисленными ключами сортируются по возрастанию, остальные располагаются в порядке создания. (Термин «целочисленное свойство» означает строку, которая может быть преобразована в целое число и обратно без изменений.)

Изображение выглядит как текст

Автоматически созданное описание

Чтобы наши целочисленные данные выводились последовательно в порядкке создания, мы можем схитрить, сделав коды не целочисленными свойствами. Добавления знака "+" перед каждым кодом будет достаточно.

Изображение выглядит как текст

Автоматически созданное описание

**Свойства и методы объекта:**

1. <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object> - все методы и свойства объектов
2. У объекта НЕТ свойства length, поэтому чтобы посчитать все пары { ключ: значение } используется счётчик и перебор – это самый простой и банальный способ
3. Object.keys(options) – данный метод берет наш объект и на его основании создаёт массив, в котором все элементы это ключи, находящиеся на первой вложенности пары { ключ: значение } у объекта. Т.к. на выходе мы получаем массив, то уже к нему можно применить свойство length, чтобы посчитать все ключи:

console.log(Object.keys(options).length); => 4

1. Object.values() – метод возвращает массив значений перечисляемых свойств объекта в том же порядке, что и цикл for in. Разница меду методом и циклом в том, что цикл перечисляет свойства и из цепочки прототипов
2. Внутри объекта мы сами можем создать через функцию метод, который будет работать внутри объекта:

Изображение выглядит как текст

Автоматически созданное описание

Запускаем новый метод: options.makeTest(); => Test

**Деструктуризация объекта:**

Деструктуризация (destructuring assignment) – это особый синтаксис присваивания, при котором можно присвоить массив или объект сразу нескольким переменным, разбив его на части.

Или же получить какое-либо свойство объекта и разбить его на части (это удобно при вложенности объекта в объекта, чтобы достучаться до значения свойства вложенного объекта). Т.е. мы вытаскиваем свойства вложенного объекта в отдельные переменные и далее можем получить их значения:



Массивы и псевдомассивы

Массивы – это частный случай объектов, структуры, которые содержат элементы по порядку. Это может быть массив с вашими закладками в браузере (именно с их адресами), или набор каких-то функций, иди даже смешанный массив с разными типами данных.

Изображение выглядит как текст

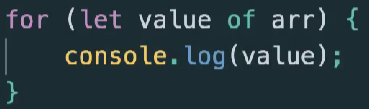
Автоматически созданное описание

**Перебор массива** может осуществляться через for:

Изображение выглядит как текст

Автоматически созданное описание

Через for of (здесь можем использовать break и continue, они важны, когда необходимо остановить цикл либо полностью, либо пропустить одно повторение):



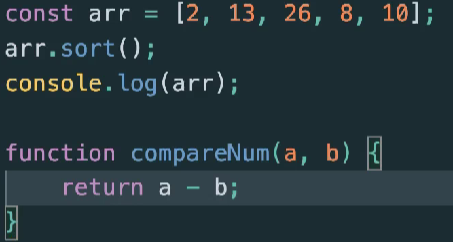
**Метод forEach** позволяет гибко перебрать все элементы, находящиеся внутри массива. Данный метод это функция, принимающая в себя callback-функцию, и эта функция будет выполняться каждый раз для каждого элемента массива. Может принимать в себя 3 аргумента: первый – тот элемент, который мы сейчас перебираем, второй – номер по порядку, третий – ссылка на тот массив, который мы перебираем. Здесь нельзя использовать break и continue, поэтому мы не сможем остановить цикл на каком-то этапе. Данный метод никогда не возвращает новый массив!

Изображение выглядит как текст

Автоматически созданное описание => Изображение выглядит как текст

Автоматически созданное описание

**Метод sort();** производит сортировку массива. Данный метод может принимать в себя callback-функцию, тогда мы сможем сказать, как именно будет сортироваться массив.

****

Это шаблон функции при сортировки массива, который довольно часто встречается! Суть данной функции в том, что она принимает в себя два аргументы, которые сравниваются. Такая функция работает, так как метод sort() внутри себя использует алгоритм «быстрой сортировки»!

**Метод Array.isArray();** возвращает true, если объект является массивом и false, если не является!

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array> - свойства и методы массива

**Псевдомассивы –** это такой объект, структура которого совпадает со структурой массива, но у таких псевдомассивов не будет никаких методов. Это просто структура, которая хранит данные по порядку.

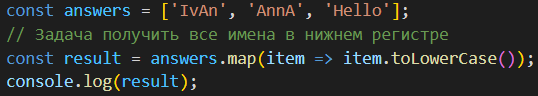
Методы перебора массивов

**Метод filter()** фильтрует элементы внутри массива. Чтобы фильтровать, нам нужно какое-то правило для фильтрации, которое прописывается в callback-функции, которая перебирает элементы и возвращает только те, что подходят под условие, внутри (). По итогу возращает новый отфильтрованный массив!

Изображение выглядит как текст

Автоматически созданное описание => [‘Ivan’, ‘Ann’]

**Метод map()** позволяет взять исходный массив и изменить каждый элемент внутри него с помощью callback-функции, которая прописывается внутри (). На выходе получим новый массив с изменёнными данными!

 => [‘ivan’, ‘ann’, ‘hello’]

**Методы every()/some()** возвращают булиновое значение(true/false).Метод *some* берёт массив, перебирает его и, если хотя бы один какой-то элемент подходит по условию, которое зададим в callback-функцию, возвращает true, иначе false.

Изображение выглядит как текст

Автоматически созданное описание => true

Метод *every* возвращает true только в том случае, если все элементы массива подходят под условие, которое прописано в callback-функции внутри ().

Изображение выглядит как текст

Автоматически созданное описание => false

**Метод reduce()** служит для того, чтобы схлопывать/собирать массив в одно единое целое. Внутри () также прописываем callback-функцию, которая принимает в себя 2 основных аргумента, которые подставляются авоматически. Первый элемент у нас обычно равен 0, а второй аргумент – это каждый элемент массива. Метод принимает также и 3-й аргумент, который является начальным значение первого аргумента, он прописывается через запятую после callback-функци.

Изображение выглядит как текст

Автоматически созданное описание => 21

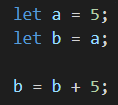
Или со строками:

Изображение выглядит как текст

Автоматически созданное описание => apple, pear, plum

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

При работе с примитивными типами данных(строки, числа, логические значения) данные внутри них передаются по значению, т.е. при изменении значения одной переменной, значение другой останется прежним:

 => b = 10; a = 5.

Но, например, при работе с объектами (объекты, массивы, функции, ошибки и т.д.) передача данных происходит по ссылке, т.е. при изменении значения ключа в одном объекте, изменится значение этого же ключа и в другом объекте:

Изображение выглядит как текст

Автоматически созданное описание => 

Такое поведение не является правильным, ведь мы внесли изменения в объект копии, а не в главный, но изменились оба. В данном случае мы не скопировали новый объект и в переменную copy не кладётся новый объект, который идентичен структуре obj. В переменную copy мы передаём значение по ссылке, т.е. данная запись передаёт не саму структуру объекта, а передаёт ссылку на уже существующий объект. Поэтому изменения происходит в двух объектах сразу, т.к. мы работаем не с копией этого объекта, а с ссылкой на структуру изначального объекта.

**Для создания копий объектов используют:**

1. Цикл

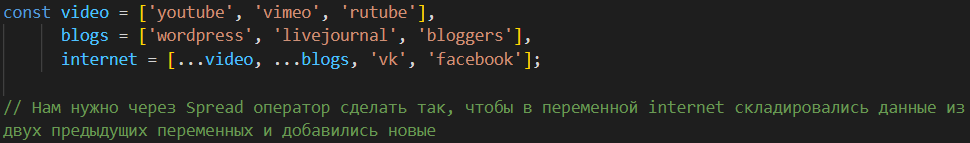
Изображение выглядит как текст

Автоматически созданное описание

Данная функция при помощи цикла создаст новый объект, перебирая старые свойства, которые были в нашем объекта.

НО! Если в начальный объект будет вложен ещё один объект, то при изменении копии вложенной структуры, она также будет меняться во всех объектах! Есть понятия глубоких и поверхностных копий объекта! При таком цикле создаётся всего лишь поверхностная копия, в таких копиях вложенные структуры также будут иметь ссылочный тип данных!

1. Метод Object.assign(objCopy, obj); Данный метод копирует все свойства объекта(ов) и помещает в новый целевой объект. Первым аргументом идёт объект, в который мы помещаем всё содержимое(целевой), вторым – тот, который мы помещаем в objCopy. В таком случае также создаётся поверхностная копия
2. Метод Array.slice(); применяется для создания **поверхностной копии массива!**
3. **Spread оператор** разворачивает структуру и превращает её в набор каких-то данных



Результат переменной internet:

Изображение выглядит как текст

Автоматически созданное описание

Также через Spread оператор мы можем раскладывать, например, те же массивы на отдельные элементы:

Изображение выглядит как текст

Автоматически созданное описание => 2 5 7

Создание копий объектов через Spread оператор:

Изображение выглядит как текст

Автоматически созданное описание

Основы ООП, прототипно-ориентированное наследование

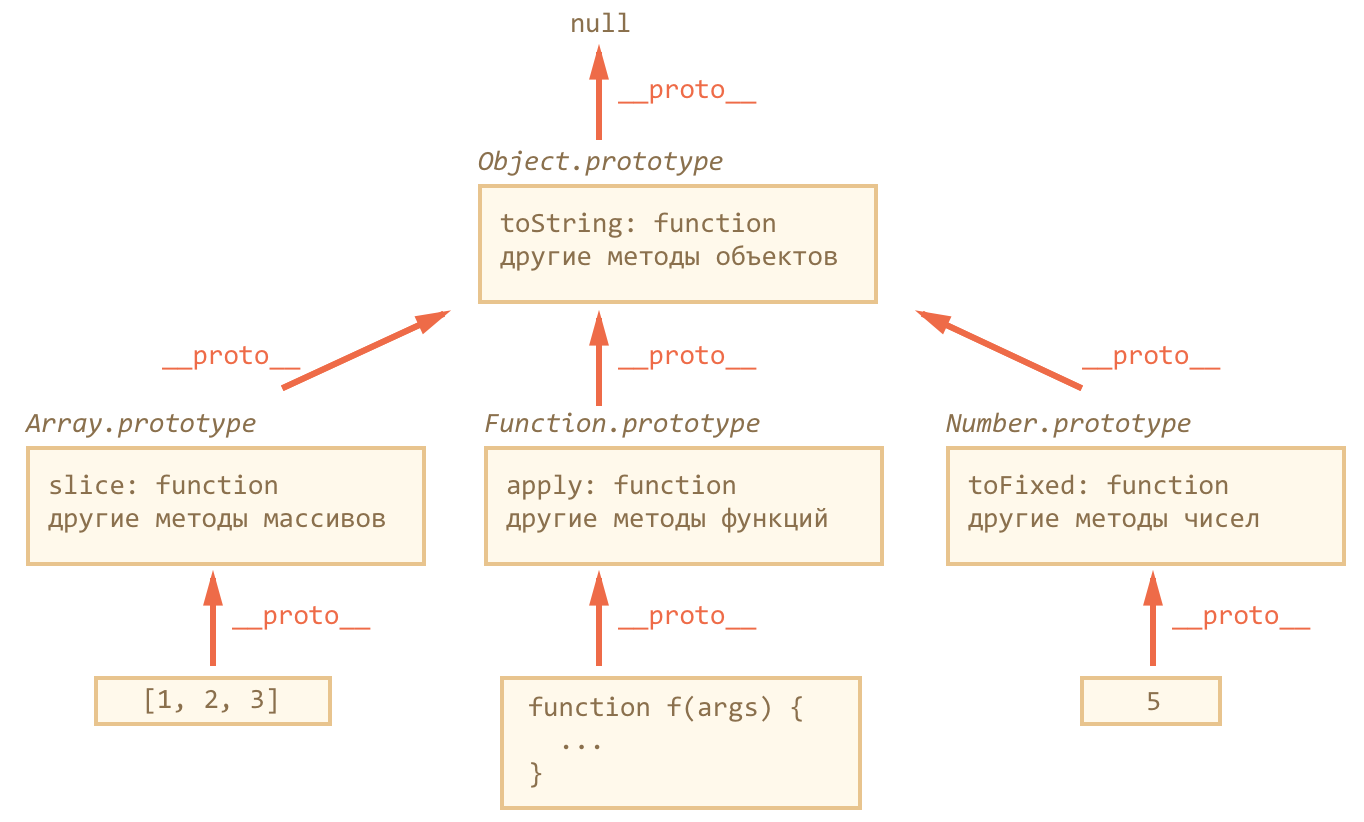
JS является объектно-ориентированным языком, главную роль в нем играет объект, он может содержать в себе методы, свойства, любой тип данных и представлять собой целостную сущность.

Смысл объектно-ориентированного программирования (ООП) заключается в том, что мы представляем любую вещь, как объект с набором свойств и методов.

Например, строка, если она записана просто в кавычки, то она примитивна, но, как только к ней применяется свойство или метод, JS оборачивает строку в объект и, после модификации, возвращает строку на место.

Когда мы, например, создаём массив, то мы создаём новый экземпляр прототипов массива. Любой экземпляр массива имеет свойство \_\_proto\_\_, ссылающееся на прототип Массива, а он, в свою очередь, исходит из главного прототипа Объекта (Объект – главная структура). То же самое происходит и с функцией, и с числом и т.д.

**Любые конструкции прототипно наследуются от Объекта!**



Устаревшее использование свойства \_\_proto\_\_ для создания прототипов. Мы установили прототипом Джона солдата. Теперь все методы и свойства, которые есть у солдата, есть и у Джона (даже функция SayHello). Также Джону мы назначили другой параметр здоровья:

Изображение выглядит как текст

Автоматически созданное описание

На данный момент на замену \_\_proto\_\_ пришли такие команды, как

1. Object.create(proto, [дискрипторы]) – создает пустой объект с заданными proto как [[Prototype]]и необязательными дескрипторами свойств

 Мы создали новый объект Джон, который будет прототипно наследоваться от солдата для того, чтобы Джон имел всё те же функции, методы и свойства, что у солдата.

1. Object.getPrototypeOf(obj) – получает прототип obj
2. Object.setPrototypeOf(obj, proto) – устанавливает прототип идентично верхней записи с \_\_proto\_\_, солдат – прототип Джона. Это работает, если у нас уже существовал Джон и мы установили прототип в динамике.

Принципы ООП:

1. Абстракция – когда отделяем концепцию от её экземпляра
2. Наследование – способность нашего объекта или класса базироваться на другом объекте или классе

Динамическая типизация

Существует понятия динамической и статической типизации. Динамическая – когда мы может один тип данных превратить в другой, статическая – наоборот.

1. Способы превращения в строку (To String)

* console.log(typeof(String(null))); => string
* Конкатенация – console.log(null + ‘’);

1. To Number

* console.log(typeof(Number(null))); => number
* Унарный плюс – console.log(typeof(+’5’)); => number
* console.log(typeof(parseInt(“15px”, 10))); 15px преобразуем в десятичную систему исчисления

1. To Boolean

* 0, “”, null, undefined, NaN => всегда false, остальные true
* console.log(typeof(Boolean(null))); => boolean
* console.log(typeof(!!“4444”));=>boolean через два знака отрицания

Замыкание и лексическое окружение

JS у каждой выполняемой функции блока кода и скрипта есть связанный с ними внутренний, то есть скрытый от нас объект, который называется лексическое окружение. Лексическое окружение делится на 2 части: на внутреннее (объект, в котором, как свойство, хранятся все локальные переменные этой функции и также некоторая другая информация(this)) и на внешнее (окружение, которое соответствует коду снаружи данной функции). Лексическое окружение существует у любого блока кода, ограниченного {}.

Каждый вызов функции это создание нового лексического окружения со своими специфичными для этого вызова локальными переменными и параметрами.

Замыкание (простым языком) – это когда функция сначала пытается найти что-то внутри себя, если она этого не находит, то обращается к чему-то, что находится более глобально.

Ключевое понятие, которое нужно запомнить, это то, что когда функция объявляется, то она содержит описание функции и замыкание. **Замыкание — это коллекция всех переменных из области видимости во время создания функции.** Функции, которые создаётся в глобальной области видимости, тоже создают замыкания. Но, так как эти функции были созданы в глобальной области видимости, то они имеют доступ ко всем переменным в глобальной области видимости. В таком случае концепция замыканий не очень уместна.

Когда функция возвращает функцию, тогда концепция замыканий становится более актуальной. Возвращаемая функция имеет доступ к переменным, которые не находятся в глобальной области видимости, но при этом существуют в её же замыкании.

<https://medium.com/nuances-of-programming/я-никогда-не-понимал-замыкания-в-javascript-часть-первая-3c3f02041970> - подробнее про замыкания!

Получение элементов со страницы. Живые и статичные коллекции

Для начала нам нужно познакомиться с понятием DOM(document object modal) – объектная модель документа. Весь наш HTML код представлен в виде дерева узлов, при этом узлы связаны между собой отношениями родительский-дочерний. После того, как документ сформирован, мы можем представить его в виде объекта, т.к. у любого объекта есть методы, мы их и будем использовать для того, чтобы иметь доступ к определённым элементам на странице.

Чтобы обратиться к DOM, нам нужно использовать такую сущность как document.method:

Более устаревшие:

1. document.getElementById(“id”); – получение элемента по его уникальному идентификатору
2. document.getElementsByTagName(“button”); – получение коллекции элементов по тегу (возвращает псевдомассив). Чтобы получить определённый элемент из коллекции, мы должны обратиться к нему по индексу (после прописи метода в конце поставить [i] и прописать нужный индекс ИЛИ уже при взаимодействии с нужным элементов прописать [i]). Даже если элемент с таким тегом на странице всего один, мы всё равно получим его в коллекции!!
3. document.getElementsByClassName(“class”); – получение элементов по классу, работает также, как получение элементов по тегам

Более современные:

1. document.querySelectorAll(“.class”); – внутрь () мы помещаем любой CSS-селектор (#ID, class и их вложенность, псевдоклассы, атрибуты и т.д.) и получаем элементы по нему. Получаем также HTMLcollection, НО в отличие от других методов, у данного метода появляется метод ForEach для перебора элементов в псевдомассиве!
2. document.querySelector(“.class”); – внутрь () также принимает любой CSS-селектор, НО возвращает только первый элемент по CSS-селектору

**Маленькое дополнение:**

- Все методы "getElementsBy\*" возвращают живую коллекцию. Такие коллекции всегда отражают текущее состояние документа и автоматически обновляются при его изменении. НО! Напрямую изменять HTMLCollection нельзя, т.е. добавлять / удалять элементы  
- Напротив, querySelectorAll возвращает статическую коллекцию. Это похоже на фиксированный массив элементов.

- Итак, когда нам нужно динамично следить за состоянием DOM-дерева, лучше использовать получение элементов через "getElementsBy\*", НО! у таких псевдомассивов НЕТ методов для работы с ними! Здесь придёт на помощь **команда, которая позволяет создать массив из массивоподобного объекта**:

Изображение выглядит как текст

Автоматически созданное описание

В таком случае мы получим уже **статичный** массив со всеми методами!

!!!Если какая-либо переменная HTML используется один раз, для получения в JS её не обязательно помещать в другую переменную, можно получить её уже при взаимодействии!!!



!!!Получать элементы можно, обращаясь не только к document, но ещё и к переменной, которая уже ранее была получена и в которой находятся элементы, которые нам нужно получить!!!



Действия с элементами на странице

Для того, чтобы понимать, что мы можем сделать с каким-то элементом на странице, можно посмотреть на этот элемент в качестве объекта через console.dir():

Изображение выглядит как текст

Автоматически созданное описание

Так мы можем увидеть все свойства, методы, стили(представлены только inline-стили, т.е. прописаны прямо в коде HTML, в CSS являются самыми приоритетными!) и т.д. данного элемента. Чтобы поменять стили элементы:

Изображение выглядит как текст

Автоматически созданное описание

**Можно назначить сразу несколько inline-стилей** через свойство cssText:



Сюда мы точно также можем подставлять какие-то переменные через интерполяцию:

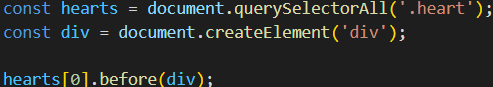


**Мы можем производить действия над множеством элементов сразу** при помощи перебирающих циклов. Можно использовать for, for…of или forEach, который действует только в методе document.querySelectorAll:

Изображение выглядит как текст

Автоматически созданное описание  
 **Методы:**

1. document.createElement(“div”); – создание нового элемента, внутрь () помещаем название того тега, который создаём. На данный момент он существует только внутри JS и на странице не появится
2. document.createTextNode(); – создаём текстовые узлы (элементы без оболочки тега), внутрь (“Hello, world”) помещаем тот текст, который хотим поместить в ноду(узел. Такой текст появится у нас на странице. Используется редко!
3. classList.(‘’); – метод classList позволяет через точку указывать, какое действие будет производиться с элементами. Чаще при работе со стилями используются не inline-стили, а работа с css классами, как здесь! В данном случае div-элемента на странице не существует 
4. document.body.append(div); – чтобы наш созданный в JS элемент появился на странице используют метод append. В данном случае мы помещаем элемент div в конец родителя body (элемент можно поместить в конец и другого родителя)
5. document.body.prepend(div); – действует обратно методу append и вставляет элемент в начало родительского элемента
6. before() и after() – методы, которые помещают элементы перед и после каких-то элементов со страницы. При записи сначала указывается элемент, перед или после которого вставится наш элемент из JS:



1. remove() – метод, удаляющий элементы со страницы



1. replaceWith() – метод, заменяющий один элемент на другой. Сначала указываем элемент, который хотим заменить, а в () указываем элемент, которым хотим заменить другой

Изображение выглядит как текст

Автоматически созданное описание

Устаревшие конструкции:

1. appendChild(div) – действует также, как метод append()
2. insertBefore(div, hearts[0]) – метод, вставляющий наш элемент div перед первым элементом hearts. Обязательно 2 аргумента! Раньше был заменой метода prepend(), before(), after()
3. removeChild() – метод, удаляющий элементы, отличается по написанию от метода remove(). Вначале прописывается родитель, далее сам метод и в () прописываем элемент, который нужно удалить
4. replaceChild() – тот же метод, что и replaceWith, НО в данном методе указывается 3 аргумента (сначала сам родитель, а после в () 1й – элемент, на который мы хотим что-то поменять, 2й – элемент, который меняется)

**Добавляем текст/HTML-код прямо в элементы:**

1. innerHTML

Изображение выглядит как текст

Автоматически созданное описание

Сначала мы создали элемент через createElement(), после поместили его на страницу в конец класса wrapper через метод append() и уже через innerHTML поместили внутрь div HTML-конструкцию

1. textContent – добавлять можно только текст! При вводе HTML-конструкции она будет превращаться в строку

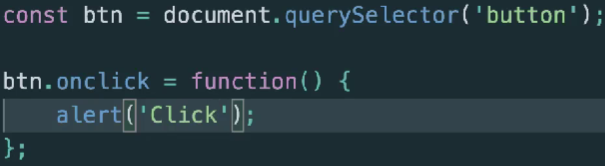
Изображение выглядит как текст, оранжевый

Автоматически созданное описание

1. insertAdjacentHTML() – метод, позволяющий вставить кусочек кода перед или после определённого тега. Сначала указывается элемент, над которым будут совершаться действия, далее прописываем метод и () прописываем 2 аргумента: 1й – спец.слово в “”, 2й – HTML-код, который мы хотим вставить также в “”. Спец.слово: beforebegin – вставляет код перед элементом, afterbegin – вставляет код в начало элемента, beforeend – в конец элемента, afterend – после элемента.

События и их обработчики

Чтобы использовать любое событие, нужно назначить обработчик события – это функция, которая срабатывает, как только событие произошло. В JS всего 3 способа назначить обработчик события:

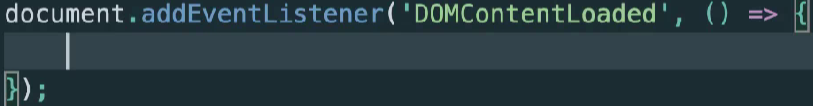
1. Использование HTML-атрибута, который записывается прямо в вёрстке, в этот атрибут мы можем передать функцию из JS для того, чтобы она выполнилась на странице. Атрибут onclick подразумевает событие click и далее идёт обработчик это функция alert. Используется крайне редко и подходит для каких-то маленьких скриптов!
2. Использование свойства DOM-дерева для события. Сначала нам потребуется получить элемент, на который мы будем вешать обработчик события. Прописываем этот самый элемент и используем на нём свойство DOM-дерево, которое называется также, как в прошлый раз, onclick и далее уже в это свойство записываем функцию, которая выполнится только после клика. Редко используется, т.к. нельзя на одно событие повесить несколько обработчиков сразу, также после использования мы не сможем удалить такой обработчик!
3. Методы addEventListener(); и removeEventListener(); Данные методы также используются на элементах, чтобы повесить на них обработчик. Первым аргументом в () мы передаём само событие, а вторым callback-функцию. Первый метод добавляет событие, а второй удаляет события.Изображение выглядит как текст

   Автоматически созданное описание

Чтобы удалить обработчик, необходимо использовать точно такую же функцию, которую мы уже назначили, поэтому лучше выносить данную функцию в отдельную переменнуюИзображение выглядит как текст

Автоматически созданное описание

Существует событие “DOMContentLoaded”, которое добавляется через addEventListenet() к document (или window). Данное событие говорит о том, что код начнёт выполняться после того, как DOM-структура будет загружена. Это событие важно использовать для того, чтобы не было ошибок. Во внутрь callback-функции помещается весь наш код. В VS code можно ввести domc + Tab и развернётся конструкция



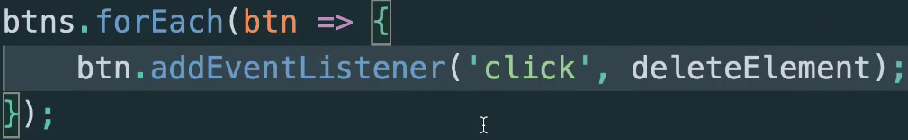
**Всплытие события** – это когда обработчик события срабатывает сначала на самом вложенном элементе и после этого поднимается наверх по иерархии нашего DOM-дерева (происходит, когда обработчики и события прикреплены на внешний и вложенный элемент, например, кнопка обёрнута в div).

**Отмена стандартного поведения браузера:**

1. Вернуть в обработчике события return: false; в самом конце (устаревший способ!)
2. Использовать метод preventDefault(), который есть у объекта события, этот участок кода помещается всегда в самое начало, так же как и обязательным условием является наличие первого аргумента event/e!Изображение выглядит как текст

   Автоматически созданное описание

Чтобы на несколько элементов сразу навесить обработчики событий, необходимо сначала полностью перебрать псевдомассив через forEach и на каждый элемент внутри навесить обработчики!



**Опции события.** У addEventListener существует и третий аргумент – это объект, в который мы можем передать различные опции (capture, once, passive, mozSystemGroup). Например, опция once указывает, что обработчик должен быть вызван только один раз, если true, то после он удалится.

<https://oddler.ru/blog/i63> - список родных событий JS

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

События на мобильных устройствах

1. touchstart – срабатывает при касании пальца по элементу на экране
2. touchmove – Каждый раз, когда палец начинается двигаться по элементу, выполняется данное событие
3. touchend – Как только палец оторвался от элемента, выполняется данное событие
4. touchenter – Оно будет срабатывать, когда мы проводим пальцем по экрану и заходим на область с элементом, на который повешено событие
5. touchleave – Событие оказывает противоположное воздействие событию touchenter – это когда палец покидает область с элементом
6. touchcancel  - Событие возникает тогда, когда точка соприкосновения больше не регистрируется на поверхности. Например, в процессе серфинга на мобильном устройстве, наш палец может выйти за пределы браузера, тогда сработает событие touchcancel.

<https://myrusakov.ru/js-touch-events.html> - подробнее о событиях с примером

**Доп.свойства у объекта события (event) при работе с сенсорными устройствами:**

* touches – свойство выдаёт список всех пальцев, которые сейчас взаимодействуют с экраном
* targetTouches – выдаёт список всех пальцев, которые взаимодействуют с нашим элементом
* changedTouches – список всех пальцев, которые участвуют в текущем событии. Например, если это событие touchend, то мы получим список с тем пальцем, который был убран, даже если остальные пальцы всё ещё на экране

Навигация по DOM - элементам, data-атрибуты, преимущество for/of:

Для получения тега body или head можно использовать document.body / document.head. Чтобы обратиться к главному тегу html используем document.documentElement, так мы получим тег html со всем его содержимым.

**Достучаться до внутренних элементов родителя** можно при использовании 3-х команд:

1. childNodes – получаем псевдомассив интересующего нас элемента с его внутренними элементами и узлами
2. firstChild – получение первой ноды родителя (есть аналог, см.ниже)
3. lastChild – получение последней ноды родителя (есть аналог, см.ниже)

**Разница DOM-элементов от DOM-узлов(нод)**: каждая сущность, которая находится на странице по факту является узлом, НО не каждый узел будет элементом! Т.е. все теги это элементы, а всё то, что возможно не видно, будет узлом (переносы строк, текстовые элементы и т.д.)

**Команды, которые позволяют получить “родителей”, “детей” и “соседей”:**

* parentNode – получаем родителя, если использовать два раза подряд, то получим элемент ещё выше, т.е. родителя нашего родителя.
* nextSibling – получаем следующую ноду
* previousSibling – получаем предыдущую ноду

Аналоги данных команд, которые не включают в себя узлы:

* nextElementSibling – получаем следующий элемент (сосед)
* previousElementSibling – получаем предыдущий элемент (сосед)
* parentElementNode – получаем элемент родителя
* firstElementChild / lastElementChild – получаем первый / последний элемент родителя
* children – аналог childNodes, так мы получаем псевдомассив только с элементами

**Создаём аналог команды childNodes, которая не учитывает узлы, через for of:** задача состоит в том, чтобы перебрать все ноды в полученном псевдомассиве и убрать текстовые ноды

Изображение выглядит как текст

Автоматически созданное описание

**Data-атрибуты –** уникальный ориентир в скрипте, это более лучший способ, чем селектор ID. Всегда начинаем прописывать data-атрибут с “data-”, а далее уже можно прописывать любое свойственное элементу название, и также мы сразу можем приписывать данному атрибуту какое-то значение, отличное от Boolean. Для получения в JS data-атрибут прописывается в [] и во внутрь прописываем полное название со значением, если прописано.

Рекурсия

**Рекурсия** – когда функция вызывает саму себя. Классическим примером выступает функция возведения в степень:

Изображение выглядит как текст

Автоматически созданное описание => Изображение выглядит как текст

Автоматически созданное описание

В первом случае мы написали функцию с циклом, во втором же случае написали рекурсию, т.к. функция внутри вызывает себя же. Т.е. мы получаем два числа: само число x и его степень n. По итогу во втором случае, если степень будет больше 1, то функция будет вызывать саму себя и чем больше будет число степени, тем больше будет цепочка вызовов.

**База рекурсии** это то значение, при котором она сразу же завершится и вернёт результат. Сама рекурсия по итогу всё равно придёт к своей базе. В примере выше базой являлась 1, при ней сразу возвращался x.

**Шаг рекурсии** – запуск вложенной функции, но уже с другим значением.

**Глубина рекурсии** – общее кол-во вложенных вызовов вместе с самым первым. В примере выше за глубину отвечала переменная n (сама степень). **Максимальная глубина рекурсии** – частный случай глубины рекурсии, обозначающий насколько максимально рекурсия может нырнуть сама в себя. Она может варьироваться от сложности действий и возможности железа, на котором запускается код (в целом максимально будет примерно 10000).

В рекурсиях не используют промежуточные переменные!

Async, defer, динамические скрипты

Атрибут defer, который прописывается в теге script при подключении js-файла, сообщает браузеру, что он должен продолжать обрабатывать страницу и загружать скриптовый файл в фоновом режиме, а затем запустить этот скрипт, когда он загрузится (данный атрибут поможет избежать ошибок, если скриптовый файл подключен не в конце html документа, а вначале тега body или в конце тега head). Скрипты с defer никогда не блокируют страницу и всегда выполняются, когда наше DOM-дерево уже готово! Срабатывает ещё до DOMContentLoaded. Благодаря этому атрибуту сохраняется последовательность выполнения всех подключённых утилит.

Благодаря атрибуту async страница не ждёт асинхронных скриптов, содержимое просто обрабатывается и отображается. Событие DOMContentLoaded и асинхронные скрипты не ждут друг друга, т.е. когда мы ставим атрибут async, скрипт начинает загружаться в фоновом режиме, но при этом запускается как только он был загружен и никого не ждёт! Остальные скрипты не ждут async, и скрипты с async не ждут остальные скрипты, они становятся независимы!

Динамически можно вставлять скрипты на страницу через js. Мы создаём новую переменную в которую помещаем новый созданный элемент с тегом script. Далее прописываем путь src и через метод append помещаем в конец тега body. Это всё можно поместить в функцию и также прописать async = false, чтобы наши динамически подключенные скрипты выполнялись последовательно!

Изображение выглядит как текст

Автоматически созданное описание

Оператор нулевого слияния (Nullish - ??) ES11

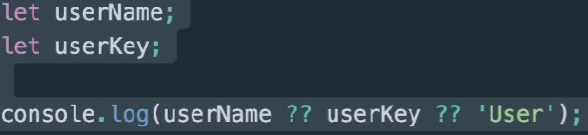
Смысл данного оператора в том, что он реагирует не на все false, а только на null или undefined.

Изображение выглядит как текст

Автоматически созданное описание => User

Этот оператор похож на оператор ИЛИ, но обращает своё внимание на то, что находится в значениях элементов и работает не со всеми 5-ю значениями false, а только null и undefined. Таким образом, мы получили выше в консоль User, потому что переданная переменная имеет значение undefined. НО, если в переменной будет значение 0, NaN, “”, false, то в консоль выведутся эти значения!

Бывает, что операция зависит от нескольких параметров и нужно, чтобы хотя бы один из них существовал. Таким образом, **из этих операторов можно выстраивать цепочку для получения первого реально существующего значения:**

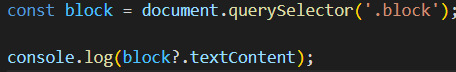


**Приоритетность Nullish оператора** находится на уровне с оператором ИЛИ (||), а это значит, что приоритет довольно низкий!

Логический оператор И (&&) и Nullish оператор НЕ могут быть смешаны!

Оператор опциональной цепочки (?.)

Смысл данного оператора в том, что он проверяет выражение слева от себя и останавливает операции, если оно имеет значение null или undefined, при этом возвращает undefined, как результат и происходит это БЕЗ ошибки!

 => undefined

Наш оператор (?) проверил значение слева от себя, и если оно имеет значение null или undefined, то оператор останавливает эту операцию (не доходит до .textContent) и возвращает undefined. В данном случае оператор опциональной цепочки(?) помог избежать ошибок в коде, т.к. изначально такого элемента с классом block на странице НЕ существует! Если бы мы не использовали данный оператор и дальше, после вывода в консоль, был бы ещё код, то у нас вылезла бы ошибка и следующий код не сработал бы!

НО! Всё это работает на чтение свойства, если попробовать записать туда что-то, то это не сработает! Дело в том, что в данном случае с элементом block, как результат мы получаем undefined, и уже в undefined мы пытаемся записать строку. Это выдаёт ошибку!



Данный оператор можно использовать и с (), и с [], НО тогда будет использоваться синтаксис полного опционального оператора (**?.**) перед скобками. Например, мы хотим вызвать в объекте метод, которого не существует:

Изображение выглядит как текст

Автоматически созданное описание

Полезные методы

1. matches() – метод, который используется на одном конкретном DOM-элементе, когда у нас есть задача среди всех элементов найти только тот, который подходит по CSS-селектору:

Изображение выглядит как текст

Автоматически созданное описание2

1. closest() – метод, позволяющий получить ближайших первых родителей по конкретному CSS-селектору (в данном случае поиск по селектору будет всё время продвигаться наверх по родителям, пока не найдёт подходящий элемент). Если элемент не будет найден, мы получим null:



Тип данных Symbol – ES6

Symbol нам необходимы для того, чтобы создавать уникальные идентификаторы, чем они и являются. Главное, что Symbol являются уникальными и неизменяемыми! Применяются к свойствам объектов:

\*В качестве названий свойств объектов используются строки (мы можем записывать названия в “”), кроме строк свойствами объектов могут быть и Symbol, только эти 2 типа данных\*

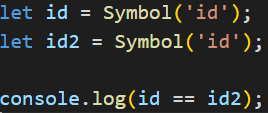
Изображение выглядит как текст

Автоматически созданное описание => получаем объект c name: ‘Test’ и symbol(id): 1

**Symbol мы можем создать БЕЗ описания** т.е. Symbol(), в таком случае мы получим свойство Symbol без описания с тем же значением 1.

Symbol важно создавать именно так, как в примере выше! Если прописать перед ним new, то это даст ошибку!

**Symbol всегда уникальны**, даже если у них одинаковое описание!

 => false

**Создание Symbol в самом объекте:**

Изображение выглядит как текст

Автоматически созданное описание

НО! Обратиться к Symbol через [id] не получиться! **Symbol позволяют создавать скрытые при обычном доступе свойства, которые ещё и не показываются при переборе объекта!** Это бывает полезно, если необходимо в объекте создать приватное(скрытое) свойство. Уникальность и скрытность такого свойства позволяет нам быть уверенным в том, что оно не будет никогда случайно перезаписано!

**Получение ссылки на Symbol:**

1. Через созданный метод в объекте

Изображение выглядит как текст

Автоматически созданное описание => 1

Создаём переменную id, в которой находится Symbol(‘id’), в объекте через [] передаём ссылку на наш Symbol и далее создаём метод по получению этого id

1. Object.getOwnPropertySymbols(obj) – метод, позволяющий получить массив Symbol и ссылку на него в нашем объекте obj, для получения их значения можно сделать так, тогда получим первое значение:

 => 1

Дескрипторы свойств и полезные методы объектов

Каким образом идёт определение того, будет ли свойство объекта показываться в циклах или нет? – У каждого из свойств объекта помимо значения, есть и специальные атрибуты, которые ещё глубже определяют, как работает это свойство. Их называют Флагами и всего их 3:

1. writable – если данный параметр будет стоять в позиции true, то свойство в объекте можно будет изменить, если стоит false – свойство будет только для чтения
2. enumerable – если данный флаг стоит в true, то свойство будет перечисляться в циклах, если false, то циклы будут игнорировать свойство
3. configurable – если стоит в true, то свойство можно будет удалить, а атрибуты его можно будет изменить, если false, то делать этого будет нельзя

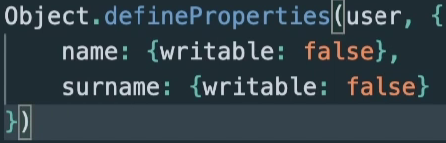
В обычной работе их не увидеть в консоли, они скрыты. Для всех создаваемых нами вручную свойств, они всегда стоят в позиции true.

**Чтобы увидеть флаги в консоли**, необходимо воспользоваться специальной командой Object.getOwnPropertyDescriptor(obj, ‘value’); Первый аргумент это объект, на котором мы используем метод, второй аргумент это свойство, флаги которого мы хотим посмотреть.

**Изменить параметры свойства** можно при помощи команды Object.defineProperty(obj, ‘value’, {writable: false}); Данный метод используется в качестве статического метода. Первые два аргумента точно такие же, как и в первой команде, а 3-й аргумент это объект, с теми флагами, которые мы хотим модифицировать. В данном случае мы поменяли флаг writable и теперь не можем менять данное свойство!

При помощи метода Object.defineProperty() мы можем создавать новые свойства, причём это свойство будет уже с заданными параметрами, которые мы укажем в 3-м аргументе, если не указывать значения флагов, то они автоматически станут false!

**Для изменения сразу нескольких свойств** объекта существует команда Object.defineProperties(obj, {values}); Первый аргумент это сам объект, а второй – объект со свойствами, которые необходимо изменить



**Полезные методы объектов:**

1. Object.preventExtensions() – предотвращает добавление новых свойств к объекту (т.е. предотвращает расширение объекта)
2. Object.seal() – запечатывает объект, предотвращая добавление новых свойств к объекту и делая все существующие свойства не настраиваемыми. Значения предоставленных свойств всё ещё могут изменяться, поскольку они остаются записываемыми
3. Object.freeze() – замораживает объект: другой код не сможет удалить или изменить никакое свойство. Объект становится эффективно неизменным. Возвращает замороженный объект
4. Object.isExtensible() – определяет, разрешено ли расширение объекта, т.е. применял ли другой разработчик метод Object.preventExtensions() на объекте
5. Object.isFrozen() – определяет, был ли объект заморожен
6. Object.isSealed() – определяет, является ли объект запечатанным(sealed)

На практике данные 6 методов используются редко

1. Object.is() – определяет, являются ли два значения различимыми (т.е. одинаковыми)
2. Object.keys() – возвращает массив из собственных перечисляемых свойств переданного объекта (перечисляемые – это те свойства, которые могут быть перечислены в цикле, и у которых флаг enumerable стоит в true)
3. Object.values() – возвращает массив значений перечисляемых свойств объекта
4. Object.entries() – возвращает массив собственных перечисляемых свойств указанного объекта в формате [key, value]

Итерируемые (перебираемые) конструкции

**Различия for…of конструкции от for…in**

1. for in при использовании на объекте, массиве или строке будет работать с каждой сущностью по порядку, которая является перечисляемой, т.е. той, у которой стоит флаг enumerable в true, и выводит в консоль именно ключи, НО данная конструкция не всегда выводит данные по порядку, поэтому не рекомендуется применять эту конструкцию на массивах и строках!
2. for of конструкция была введена в ES6 стандарте, она проходится по значениям перебираемого объекта и делает с ним то, что мы укажем. Выводит именно значение, а не ключ. Конструкцией for of нельзя перебрать объект!

**Перебираемый или итерируемый объект** – это тот, который мы можем использовать в for of. Строка и массив это итерируемый объект, а объект нет. В свою очередь это определяется тем, есть ли у этой сущности Symbol.iterator (это встроенный технический символ, который говорит о том, что сущность перебираема). Наглядно его можно увидеть в консоли браузера, если выведем какую-то сущность, как объект через condole.dir();

**К перебираемым сущностям относятся**: массивы, строки, типизированные массивы, map, set и DOM-коллекции.

Мы получаем 2 преимущества: 1-е это строгое соответствие порядка в переборе, 2-е это то, что идёт исключение всех лишних свойств, которые могут быть унаследованы (например, у нас не будет выводится какой-то зарезервированный внутри сущности метод)

.iterator – это метод, который возвращает объект с методом next. Когда мы хотим сделать объект перебираемым, то .iterator приходится создавать вручную. Чтобы сделать объект перебираемым (итерируемым), т.е. по простому for of позволить работать с ним, нам необходимо добавить в объект метод с именем Symbol.iterator:

Изображение выглядит как текст

Автоматически созданное описание

Мы обращаемся к объекту obj и открываем [], в которых будем создавать новый Symbol.iterator, который является техническим, готовым в JS. Далее нужно создать метод, который отвечает за этот перебор, поэтому прописываем функцию. Когда наш цикл for of будет запускаться на объекте obj, то он вызывает этот метод, который мы прописали, всего 1 раз и этот метод должен нам вернуть итератор, это объект с методом next. И дальше for of будет работать уже с этим объектом, который вернулся их этого метода. For of получил наш объект с методом next, и теперь когда наш перебор хочет получить следующее значение, он вызывает метод этого объекта (next) и результат вызова этого метода должен иметь определённый вид. Это будет объект со свойством done (контролирует работу цикла, если стоит в true, то цикл завершён, если в false, то ещё работает) и value (второе прописывается не всегда, а только тогда, когда done: false, т.к. цикл ещё работает и обязательно нужно вывести какое-то значение, с которым дальше цикл будет работать).

Map (карты)

Картами или Map-ами называются специфические структуры данных, которые очень похожи на объект, только у них вместо свойств могут использоваться и объект, и массив, и функция и т.д. (в объекте свойствами могут быть только строки и символы).

**const map = new Map();**

Создаём новую переменную и в неё помещаем новую Карту, во внутрь () можно передавать начальное значение:

Изображение выглядит как текст

Автоматически созданное описание

Изначально карта пустая и представляет собой внутри массив массивов! Map – это итерируемый объект, к которому можно применять for of даже без ручного добавления Symbol.iterator! Порядок свойств в Map всегда такой, в как мы добавляли, в отличие от объектов, в которых нет чёткого порядка! **У Map есть свои методы:**

1. Set() – сначала обращаемся к Карте, на которой хотим этот метод применить через точку прописываем set(). В () передаём 2 аргумента: ключ и значение.

В примере ниже мы получаем объект, у которого первым свойством (ключом) является объект

Изображение выглядит как текст

Автоматически созданное описание => 

Команду set можно выстраивать в цепочку:

Изображение выглядит как текст

Автоматически созданное описание

Но это не слишком рационально, если таких элементов будет много, поэтому можно воспользоваться циклом или методом ForEach:

Изображение выглядит как текст

Автоматически созданное описание => Изображение выглядит как текст

Автоматически созданное описание

1. get() – получение каких-то данных из Map



1. has() – проверка на наличие чего-то внутри Map



1. delete(key) – удаляет что-то из Map
2. clear() – полностью очищает наш Map
3. size – это свойство, которое возвращает кол-во элементов на данный момент внутри Map

**У Map существует 4 способа перебора значений:**

1. keys() – метод, который возвращает итерируемый объект по ключам

Изображение выглядит как текст

Автоматически созданное описание => Изображение выглядит как текст

Автоматически созданное описание

1. values() – метод, возвращающий итерируемый объект по значениям

Изображение выглядит как текст

Автоматически созданное описание => Изображение выглядит как текст

Автоматически созданное описание

1. entries() – метод, который возвращает объект в виде массива, который содержит подмассивы по два элемента внутри

Изображение выглядит как текст

Автоматически созданное описание

Также в цикле for of 1-ю переменную можно прописать как массив и после получить свойство и значение по отдельности

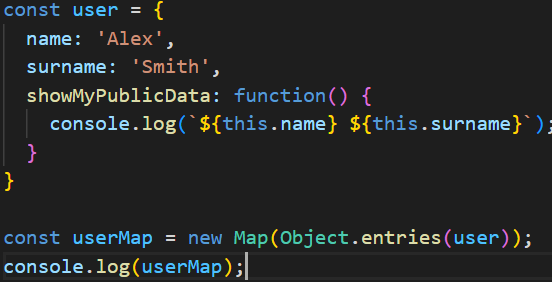
Изображение выглядит как текст

Автоматически созданное описание => Изображение выглядит как текст, электроника, клавиатура

Автоматически созданное описание

1. forEach(value, key, map) – метод перебора Map, в котором прописываются 3 аргумента: значение, ключ и сам Map

**Способы перевода Объекта в Карту и наоборот:**

 => Изображение выглядит как текст

Автоматически созданное описание

И обратно в Объект:

** => Изображение выглядит как текст

Автоматически созданное описание**

Set

Set – это специальная структура данных, подразумевающая под собой особый вид коллекций по типу массива, где каждое значение не имеет ключа и может повторяться только один раз.

Set работает оптимизированнее и быстрее, чем обычный массив!

**const set = new Set();**

**Методы Set:**

1. add() – метод, который добавляет новое значение в массив Set (если данное значение уже существует в массиве, то оно повторно добавлено НЕ будет). Добавлять новые уникальные значения можно по цепочке, как в примере с Map ↑
2. delete(value) – метод, который удаляет какое-то значение из массива Set
3. has(value) – метод, проверяющий наличие какого-то значения в Set
4. clear() – метод, очищающий полностью массив Set
5. size – свойство, которое возвращает размер Set

**Варианты для перебора Set:**

1. for of => **for (let value of set) console.log(value);**
2. forEach(value, valueAgaing, set) – 3 аргумента: значение, ещё одно значение, оно тоже самое, что и первое и создано лишь для того, чтобы была совместимость с данным методом перебора, и сам Set
3. values() – метод выводит значения
4. keys() – метод, который нужен лишь для обратной совместимости с Map, т.к. в Set ключей, как таковых, нет (выводит тоже, что и values())
5. entries() – существует также лишь для обратной совместимости с Map

Часто **с помощью Set реализовывают функцию-помощник**, которая фильтрует любой массив:

Изображение выглядит как текст

Автоматически созданное описание

Т.е. мы возвращаем массив из переданного массива, который помещён в Set. Таким образом, происходит быстрая фильтрация: массив передаётся в Set, где избавляется от повторяющихся элементов, а после формируем обратно тот же массив, который уже будет возвращён из данной функции только уже с уникальными данными!

BigInt

Максимально возможное число, с которым JS может работать записано в Number.MAX\_SAFE\_INTEGER(); => 253-1

Если нужно использовать числа выше, то необходимо применять такой тип данных, как BigInt, чтобы избежать ошибок при работе с такими большими числами!

Чтобы создать тип данных BigInt, необходимо в конце числа прописать букву n:

**const bigint = 13564244256376389959857646535n;**

Второй вариант это использовать напрямую команду, которая будет создавать BigInt:

**const sameBigInt = BigInt(13564244256376389959857646535);**

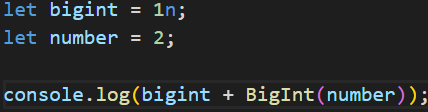
Данная команда принимает внутрь себя также и строки!

1. BigInt НЕЛЬЗЯ использовать с методами со встроенным объектом Math
2. НЕЛЬЗЯ смешивать в операциях BigInt и обычные числа

**С BigInt можно использовать операторы:** сложение, вычитание, умножение, возведение в степень и деление с остатком, побитовые операторы и операторы сравнения. Чуть сложнее будет использовать деление с большими числами, т.к. операция деления ВСЕГДА будет возвращать округлённое значение без дробной части!

При использовании операторов сравнения они будут нормально работать даже если смешивать BigInt с обычными числами! И т.к. это 2 разных типа данных, то равны они будут только при Нестрогом сравнении (==). Например, 2n == 2 это true!

Если необходимо сложить обычное число с BigInt, то придётся явно конвертировать типы данных:

 => 3n

Или наоборот:

 => 3 (через унарный плюс НЕ сработает)

НО в данном случае при переводе BigInt в обычное число нужно быть осторожными, т.к. если число BigInt больше 253-1, то часть числа будет отброшена до допустимой границы!

ClassList и делегирование событий

**Свойство classList** довольно большое и содержит как метода, так и полезное **свойство length**, которое возвращает кол-во классов у элемента!

**Методы свойства ClassList:**

1. item(index) – метод повзоляет получить класс, который распологается под определённым индексом
2. add(‘class’, ‘class2’) – метод добавляет новый класс к элементу (можно передавать несколько классов, как и в методах ниже)
3. remove(‘class’) – метод удаляет определённый класс у элемента
4. toggle(‘class’) – метод добавляет класс, если его нет, и убирает, если он есть
5. contains(‘class’) – метод позволяет проверять наличие какого-то класса на определённом элементе и возвращает boolean значение

**Делегирование событий** – отлично подходит, если есть много элементов с одинаковыми обработчиками событий, причём при динамическом изменении они точно также будут применятся к новым элементам. Суть его в том, что мы берём элемент, который является родителем для, например, всех кнопок и работаем уже непосредственно с ним, т.е. мы ему назначаем обработчик события клика, а внутри кода уже будем проверять, на какой элемент кликнули. Так мы назначаем функцию для его потомков, если они подходят под какие-то определённые параметры. Мы делегируем событие с родителя на его потомков.

Изображение выглядит как текст

Автоматически созданное описание

Мы вешаем обработчик события клика на элемент wrapper и далее callback-функция с аргументом event (объект события), который содержит в себе всю информацию о том элементе, на котором происходит событие! После проверяем условием, что мы кликнули на определённый элемент, в данном случае это button. Также мы проверяем наличие самого event.target, т.к. в HTML-структуре не все элементы поддерживают событие клика (например, тег br невозможно кликнуть), поэтому это важно учитывать, чтобы избежать ошибок! Теперь, когда мы будем кликать на button, в консоли будет Hello.

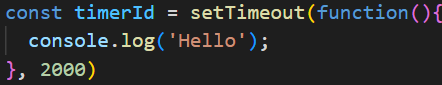
Можно также использовать event.target.matches(“button.class”) метод вернёт true или false, в зависимости от того, соответствует ли элемент указанному css-селектору.

Скрипты и время их выполнения. setTimeout и setInteval

setTimeout – конструкция запускает определённый участок кода через какое-то время. Синтаксис:

**const timerId = setTimeout();**

У этой функции такой синтаксис, что сначала она принимает ту функцию, которая должна будет запуститься через определённый промежуток времени. Причём она принимает либо объявление функции, либо её название, она не вызывается прямо здесь и сейчас! Время, через которое функция вызовется, мы задаём вторым аргументом в милли секундах (1000 мсек = 1 сек). Можно также передавать третий аргумент, который будет относится к внутренней анонимной функции (аргументов может быть много):

 ⬄ Изображение выглядит как текст

Автоматически созданное описание

Можно также передавать уже готовую функцию в setTimeout:

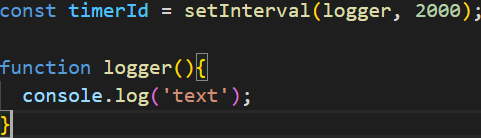
Изображение выглядит как текст

Автоматически созданное описание

**Функция setTimeout НЕ обязательно помещать в переменную**, она будет работать и без неё. **НО** когда мы передаём setTimeout или setInterval в переменную, мы записываем в неё числовой идентификатор этого таймера, делается это для того, чтобы можно было чётко определять различные setTimeout, потому что таких ассинхронных ф-ций может быть много, а определять их нужно, чтобы потом останавливать!

**Останавливать setTimeout может clearInteval(timerId);** во внутрь которого помещает уникальный идентификатор таймера, она же переменная.

**setInterval –** функция, которая задаёт временной интервал повторений функции таймера. Выглядит точно также, как setTimeout и аргументы принимает такие же, отменяется точно также. Теперь каждые 2 сек в консоль будет выводится text:



**Чем рекурсивный setTimeout лучше, чем setInterval?** – когда таймер с интервалом (setInterval) работает, он не учитывает, как долго будет работать функция внутри него! Т.е. функции setInterval всё равно, сколько внутри будет выполняться функция logger. Для решения этой проблемы используют рукурсивный setTimeout.

Изображение выглядит как текст

Автоматически созданное описание

Аргумент delay в setTimeout(function, delayTime) не означает точное время задержки, после которого функция выполнится. Он означает минимальное время ожидания, после которого в какой-нибудь момент времени, функция будет вызвана.

Сборщик мусора и утечка памяти

**JavaScript – это высокоуровневый язык** программирования, т.к. многие базовые операции уже реализованы за нас на программном уровне, например, работа с оперативной памятью компьютера.

**JavaScript – это интерпритируемый язык** программирования, а не компилируемый, т.е. программа интерпритатор построчно запускает JS код и будет выполнять его, будь то в браузере или в терминале.

В высокоуровневых языках работа с памятью происходит условно автоматически, т.е. интерпритатор сам решает, когда выделить память, как её использовать, и как её освободить. НО на этапе освобождения памяти могут возникнуть проблемы, именно поэтому говорим условно автоматически, т.к. задача точного определения, что эта часть памяти больше не нужна, она как бы алгоритмически однозначно неразрешима, поэтому иногда сам разработчик должен решать эти вопросы и помогать интерпритатору, иначе могут быть утечки памяти!

**Сборщики мусора** – это некие подпрограммы , которые отслеживают выделение и использование памяти компьютера, и при необходимости они могут автоматически освобождать эту память, если какой-то код сейчас не используется, он не является достижимым, то есть нет входящих ссылок. Если ссылок нет, то эта часть кода уже не нужна и будет удалена из памяти, НО это может также привести к утечкам памяти!

**Случаи утечек памяти:**

1. С глобальными переменными, если в коде не прописан строгий режим (‘use strict’). В таком случае можно создать перемнную без её объявления (let, const), в коде это значит, что мы создаём лишнюю глобальную переменную, которая помещается в глобальный объект window, и такая глобальная перемнная не может быть удалена сборщиком мусора!
2. Забытые таймеры. Проблема в том, что эти таймеры хранят ссылки на то, с чем мы должны работать. Т.е. например, был создан таймер, в котором хранятся ссылки на переменную с элементом из DOM-дерева, но если этот элемент будет удалён со страницы, то он всё равно останется в памяти, т.к. ссылка на него будет существовать в таймере, который срабатывает, например, каждую секунду. Так произошла утечка памяти, т.к. память забивается ненужными данными. Важно останавливать ненужные таймеры!
3. Обработчики событий на несуществующих элементах. Т.е. элемент со страницы может быть удалён, а его обработчик нет, и в нём же остаётся ссылка на элемент, с которым нужно работать. НО теперь это не проблема, т.к. современные браузеры автоматиески будут удалять обработчик события, если элемент, на который они повешены, был удалён!
4. Замыкание. Теоретически можно создать случай, когда получится утечка памяти за счёт того, что большой объем данных хранится в замыканиях функций и не может быть удалён.
5. Ссылки на DOM-элементы. Очень часто ссылки на них сохраняем в перемнных JS, одновременно с этим эти элементы существуют в структуре DOM. Когда мы удаляем объект из DOM-дерева, то ссылка на него может оставаться в перемнной JS, и этот элемент будет оставаться в памяти.

WeakMap(слабая карта) и WeakSet(слабый набор)

Это специальные структуры данных, которые используются реже, чем Map и Set, и служат, в основном для оптимизации кода. Они являются доп. хранилищем данных для объектов, управляемых из каких-то других мест в коде. Эти структуры данных контролируют использование памяти.

**У WeakMap есть 2 особенности:** в ней ключами могут быть только объекты, и, если нет ссылки на этот объект и он существует только внутри WeakMap, то этот объект будет удалён из WeakMap. Таких методов, как keys, values, entries у WeakMap не существует, также как и цикла перебора for of. Доступны только set, get, delete, has.

**Коллекция WeakSet** аналогична обычному Set, но мы можем добавлять в WeakSet только объекты, не примитивные назначения! Объект присутсвует в множестве только до тех пор, пока он где-то доступен ещё, т.е. есть ссылка на него, иначе будет удалён. Также WeakSet поддерживает такие методы, как add, has и delete. WeakSet НЕ является перебираемым! Все такие ограничения связаны с тем, что это динамическая структура, которая может быстро очищаться сборщиком мусора, также и у WeakMap.

Работа с датами

Дата – это специфический объект, который содержит в себе различные свойства и методы. Дату можно использовать для измерения времени работы какой-то функции, если мы хотим засечь, сколько она работает, и сравнить с другой функцией. Также можно использовать для вывода текущей даты, это может быть таймер или часы на странице или для хранения времени создания или изменения какого-то документа.

**const now = new Date();**

В () можно передавать аргументы. Например, мы можем передать дату в виде строки, это часто используется, когда на странице есть input, у которого стоит type: date (HTML5):

 => 

**Можно чётко настраивать нашу Дату**, передавая чёткие параметры: в таком же порядке передаём год, месяц(отсчёт идёт от 0, т.е. 0 - январь), день, часы(учитывают часовой пояс, т.к. у меня GMT+3, то 20 превратится в 17 по Гринвичу), минуты, секунды и миллисекунды(все Даты хранятся в миллисекундах, это значение называется timeStamp и отсчёт начинается от 1970г, чтобы получить дату раньше, нужно передать отрицательное значение):

 =>   
 **Методы получения компонентов Даты,** они возвращают значения в соответсвии с местным часовым поясом:

1. getFullYear() – получение текукщего года
2. getMonth() – получение текущего месяца, отсчёт от 0
3. getDate() – получение текущего дня месяца от 1 до 31
4. getDay() – получение текущего дня недели, нумерация от воскресения это 0 день, а 6 день это суббота и т.д.

Для получения другого часового пояса используются аналогичные методы с UTC:

1. getUTCHours() – получение текущего часа по Гринвичу

Другие методы:

1. getTime() – возвращает timeStamp(содержит число миллисекунд прошедших с 1 января 1970 г UTC).
2. getTimezoneOffset() – метод получет разницу в минутах между часовым поясом и UTC

**Методы set, т.е. мы можем установить дату:**

1. setHours(20, 40) – метод устанавливает в текущей дате 20 часов, но также можно установить и минуты вторым аргументом. При работе с Датами существует автоисправление, т.е. если мы зададим первый аргумент в 40 часов, то нам будет корректно выведена новая Дата.

**Метод parse:**

1. new Date.parse(‘2023-01-08’) – метод возвращает миллисекунды переданной даты. НО по идее время будет показываться с ошибкой, тк в случае если строка дана в формате ISO ('2021-06-26') вернется время равное UTC. Чтобы cчитало корректно - формат даты должен выглядеть так: 'June 26, 2021'

Параметры документа, окна и работа с ними

document – это объект, который содержит всю нашу HTML-структуру, только в более сложном варианте

window – глобальня сущность, через которую видно весь документ/сайт, по факту весь видимый экран. К window относятся методы:

1. scrollBy(x, y) – скролит страницу относительно текущего положения, аргументы передаются по координатам высоты и ширины, т.е. насколько будет проскролена страница.
2. scrollTo(x, y) – скролит страницу относительно всей страницы, аргументы такие же, как у scrollBy()

screen – по факту весь видимый монитор

Изображение выглядит как стол

Автоматически созданное описание

getBoundingClientRect() – метод возвращает все координаты элемента (в JS координаты рассчитываются от верхнего левого угла!)

window.getComputedStyle(element, psevdoelement) – метод возвращает CSS-стили, которые были применены к элементу. Позволяет узнать о применении какого-то определённого стиля. Изменять стили через данный метод нельзя! Через этот метод можно получить стили псевдоэлемента, который относится к нашему элементу, для этого псевдоэлемент стоит прописать как второй аргумент в (), после нашего главного элемента!

MutationObserver, ResizeObserver и contenteditable

MutationObserver – встроенный объект, который отслеживает изменения самих элементов. Срабатывает уже после изменений, т.е. нельзя что-то сделать во время изменения, работа происходит уже с результатом. Например, нельзя запретить писать определённые буквы. Также это всё ассинхронная операция, т.е. отслеживание может выполнится чуть позже или чуть раньше, в зависимости от различных условий.

**Изображение выглядит как текст

Автоматически созданное описание**

Создаём сущность, которая будет следить за элементом box. Во внутрь () передаём callback-функцию, которая будет происходить, после того, как произойдёт какое-то изменение элемента. Аргумент mutationRecords. Объект, который будет следить за чем-то, мы создали, теперь необходимо сказать у какого определённого элемента отслеживать изменения и выполнять функцию в случае изменений. С этим нам поможет метод observe, в котором 2 аргумента: сам элемент, за которым нужно следить, и config, т.е. определённые настройки что конкретно нужно отслеживать. Если нам уже не нужно отслеживать элемент, то нам поможет метод disconnect();

contenteditable – атрибут, который используется довольно редко, но позволяет включать редактирование страницы пользователем

ResizeObserver – работает по такому же принципу, что и MutationObserver, только отслеживает изменение размеров элементов.

Функции-конструкторы

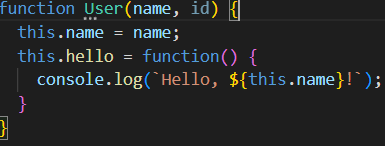
Если функция будет содержать методы и свойства, то она в итоге создаст новый объект, т.е. созданием новой функции мы создаём новый объект, свойства которого уже прописаны в этой функции!

Изображение выглядит как текст

Автоматически созданное описание => 

Создаём функцию User, в которую запишем свойства, которые будут отображаться также, как в объекте. Ключевое слово this помогает создать name и id уникальные для каждого конкретного пользователя. Когда мы использовали такой синтаксис, наша функция стала конструктором, с её помощью можно создавать теперь новых пользователей. И в новой перменной lida будет находиться не функция, а объект, потому что функция User стала конструктором, теперь, когда она вызывается при помощи ключевого слова new, она создаёт новый объект с теми свойствами, который мы ей прописали в функции. В таких функциях нам не нужен return, мы ничего из них не возвращаем!

Кроме свойств в функции-конструкторы можно также прописать методы. Прописав один раз метод прямо в нашем прототипе, он будет у каждого из наших потомков:



При помощи свойства prototype можно добавлять новые методы или свойства в наш конструктор, и они будут прототипно наследоваться у потомков. Этот приём используется, когда у нас нет доступа к нашему прототипу, либо мы не можем менять его по каким-то причинам, но его нужно немного модифицировать, т.е. добавить какие-то доп. плюшки:

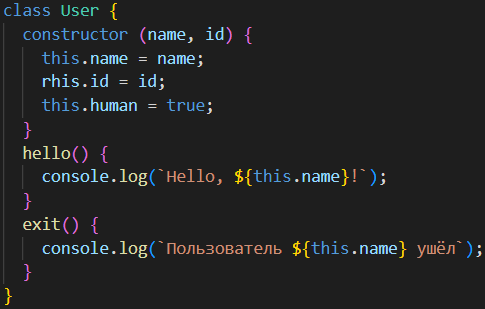
Изображение выглядит как текст

Автоматически созданное описание

Теперь прототипно будет наследоваться и этот метод, но он появится у всех потомков, которые созданы/написаны после него

Функции-конструкторы необходимы для создания новых однотипных объектов, в практире это могут быть новые пользователи сайта, товары в магазинах, ролики на ютубе, везде где есть шаблонизация.

В стандарте ES6 появились Классы – это так называемый синтаксический сахар, т.е. красивая обёртка всего выше описанного функционала, который существует внутри, но Классы действительно удобнее использовать!



Констекст вызова. This

Если говорить по простому, то Контекст вызова – это то, что окружает функцию и в каких условиях вызывается. Функция может вызываться 4-мя способами, и в каждом Контекст вызова может отличаться!

1. Обычная функция. В таких функциях Констекст вызова(this) будет ссылаться на глобальный объект Window, если НЕ установлен строгий режим(“use strict”). Если установлен строгий режим, то *this* превратится в undefined!

Изображение выглядит как текст

Автоматически созданное описание

Задача для собеседований. Вопрос в том, что выведет в консоль контекст вызова функции sum и вернётся ли сумма a и b? Выведет undefined или window в зависимости от режима, а вот сумма вернётся в том случае, когда будет замыкание, т.е. нужно убрать this и тогда при запуске функции sum, она будет искать эти переменные сначала внутри себя, а после поднимется выше и обращается к родительской функции, которая имеет доступ к двум аргументам, которые после были переданы.

1. Метод объекта. В консоль получаем этот же объект, в котором находится наш метод. Если мы используем метод внутри объекта, то Констект вызова всегда будет ссылаться на этот же объект.

Изображение выглядит как текст

Автоматически созданное описание => 

1. Классы или Функции-конструкторы, использование функций через оператор new. Внутри таких функций Констект вызова для всех методов и свойств будет только что созданный новый объект. This всегда ссылается на только что созданный новый объект! Здесь *this* это новый экземпляр объекта!
2. Ручное присвоение *this* любой функции. Методы *call / apply*!

Изображение выглядит как текст

Автоматически созданное описание => Изображение выглядит как текст

Автоматически созданное описание

Мы создали функцию sayName, которая выводит сам Констект вызова и *this.*name, т.е. какое-то имя должно вывестись в консоль. Также у нас будет существовать какой-то user, это будет объект со свойством name. Задача в том, как сделать так, чтобы функция, когда запускалась, она брала как Контекст вызова НЕ window/undefined, а именно user, чтобы можно было получить доступ к его name? Для этого существует методы *call / apply*, во внутрь которых передаём Контекст вызова, который хотим передать в функцию!

**Разница в методах call / apply** начинается тогда, когда функция принимает в себя доп.аргументы. В случае с call эти аргументы передаются внутри () через запятую. В случае с apply аргументы необходимо передать в массиве:

Изображение выглядит как текст

Автоматически созданное описание => Изображение выглядит как текст

Автоматически созданное описание

**Метод bind** создаёт новую функцию, связанную с определённым Контекстом, в предыдущих методах функция новая не вызвалась, просто подвязывался Контекст

Изображение выглядит как текст

Автоматически созданное описание => 6

Мы создали функцию count с агрументом num, в которую также должен передаваться Контекст вызова. Функция count должна удваивать все числа, записанные в аргумент num. С этим функционалом нам поможет метод bind, который помещаем в отдельную переменную и присваиваем значение 2, т.к. метод должен удваивать. Благодаря методу bind в переменную double помещается новая функция, которая принимает в себя аргумент num и удваивает его. Вот и всё!

**У стрелочных функций нет своего Контекста вызова!** Они всегда будут брать Контекст вызова у своего родителя:

Изображение выглядит как текст

Автоматически созданное описание => 

В данном случае как результат работы стрелочной функции, помещённой в переменную say, и вывода в консоль его Контекст вызова, мы получаем сам объект obj, т.к. стрелочная функция обращается к Контексту своего родителя – метода sayNumbers, Контекст которого это сам объект, в котором этот метод находится!

Стрелочные функции часто используются для модификации каких-то элементов прямо здесь на месте и есть ещё синтаксические нюансы. Если действие в стрелочной функции помещается в одну строку, то его можно прописать без {} и без return! Также если она принимает только один аргумент, то он может быть записан без ().

Когда у нас в обработчик события передана callback-функция в виде обычной функции, то Контекст вызова *this* это сам элемент, на котором произошло событие или практически тоже самое, что event.target, который используется чаще! НО если передана стрелочная функция, то Констекст потеряется и будет обращаться или к undefined или к window.

Классы (ES6)

**Классы** – это красивая обёртка функций-конструкторов или “синтаксический сахар”. Классы, как и функции-конструкторы служат для создания новых объектов.

**Синтаксис:** начинаем с ключевого слова class, затем прописываем с большой буквы название класса и открываем {}. Чтобы сконструировать наш класс прописываем constructor(аргументы, приходящие извне) {}. Когда переданы аргументы, их нужно записать в свойства нового объекта, сделать это можно через Контекст вызова *this*, через который мы будем обращаться к экземпляру нового созданного объекта, в который записываем свойства! Метод создаётся отдельно вне constructor, между методами и constructor не ставится ; ! Теперь когда наш метод готов и основные свойства заданы, готовый Класс можно использовать. Создаём новую переменную, в которую помещаем объект, который создаётся при помощи Класса через ключевое слово new!

Изображение выглядит как текст

Автоматически созданное описание

**Можно наследовать Классы**, для этого есть ключевое слово extends! А чтобы внутри нового Класса, который наследуется от другого Класса не повторять его свойства, используется super(); которая вызывает суперконструктор родителя, т.е. вызывает тоже самое, что было у родителя (свойства и методы), и оно должно стоять на первом месте в конструкторе!

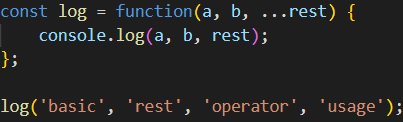
Изображение выглядит как текст

Автоматически созданное описание

Extends передаёт все методы по умолчанию, а свойства из constuctor - только по выбору (посредством super())!

Rest оператор и параметры по умолчанию(ES6)

**Rest(оставшиеся элементы)** – это брат оператора разворота (Spread), который использует точно такой же синтаксис, но уже в других условиях. Если Spread брал сущность и раскладывал её на отдельные элементы, то Rest занимается обратным: отдельные элементы объединяет в один массив.

 => 

В данном примере мы передали Rest оперетор в качестве аргумента к функции, т.к. точно знаем, что к нам придут переменные *a* и *b*, а сколько дальше переменных будет мы не знаем. Важно! Rest оператор всегда прописывается в конце, его синтаксис – это три точки(…), название переменной Rest оператора можно задать любое! С аргументами *a* и *b* также были переданы ещё два аргументы, которые благодаря Rest оператору сформировались в один массив.

**Параметры по умолчанию –** позволяют дать аргументу функции значению по умолчанию, т.е. если при вызове функции аргумент не будет передан, то значение подставится автоматически то, которое мы указали в самой функции:

**Изображение выглядит как текст

Автоматически созданное описание** => 2

AJAX и общение с сервером

Чтобы страница умела общаться с сервером, необходимы http-запросы, которые будет отправлять. Мы может запрашивать данные, постить и выполнять другие операции, и чтобы всё это происходило ассинхронно, без перезагрузки страницы, нам нужна технология AJAX(Asynchronous Javascript and XML).

Реализуется несколькими способами:

1. При помощи объекта XMLHttpRequest – на данный момент не актуален. Создаётся через спец.слово new – это конструктор, который создаёт новый объект, у которого есть свои методы, свойства и события.

Методы:

* open(method, url, async, login, pass) – собирает настройки, которые помогут в будущем сделать запрос. Принимает в себя аргумент method(используется для запроса: GET, POST…), url(путь к серверу), async(ассинхронность, стоит в true), login и pass(некоторые запросы можно делать при наличии логина и пароля).
* setRequestHeader() – http-заголовки
* send() – отправка запроса на сервер

Свойства:

* status – статус нашего запроса
* statusText – текстовое описание ответа от сервера
* response – ответ от сервера, т.е. то, что задал бэкенд-разработчик
* readyState – содержит текущее состояние нашего запроса

События:

* readystatechange – отслеживает статус готовности нашего запроса в данный момент (следит за свойством readyState)
* load – срабатывает, когда наш запрос полностью загрузился и мы получили какой-то результат
* loadstart – начало нашей загрузки
* progress
* timeout
* loadend

Когда используем XMLHttpRequest в связке с FormData, заголовки указывать не надо, это приведёт к ошибке!!!

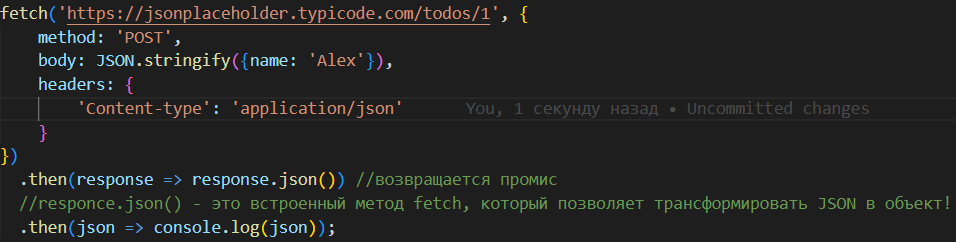
1. **FetchAPI**(Application Programmic interface, т.е. интерфейс какого программного обеспечения/приложения) – это набор данных и возможностей, который предоставляет нам какое-то готовое решение. Данная технология помогает общаться с сервером и построена на Промисах.

**Использование:** прописываем ключевое слово fetch, открываем () и внутрь помещаем тот URL, на который будем посылать запрос, если больше ничего не указывать, то это будет классический GET-запрос, который просто получит данные по этому URL. Конструкция fetch возвращает Промисы, которые можно обработать при помощи цепочки then!

**Другие запросы:** Чтобы использовать запросы POST, PUT и т.д., нужно добавить настройки, после URL, в виде объекта, который содержит основные настройки: method и body, т.е. тело, которое будем отправлять, также лучше всегда указывать headers(заголовки), которые будет определять, какой контент мы отправляем

**Методы:**

* json() – позволяет быстро трансформировать JSON в объект JS
* text()



Важная особенность fetchAPI: Промис, который запускается при помощи fetch НЕ перейдёт в состояние reject(отклонено) из-за ответа HTTP, которое считается ошибкой(404, 500, 502 и т.д.), он всё равно выполнится нормально, единственное, что поменяется – это свойство status, которое перейдёт в состояние false! Reject будет возникать в случае сбоя сети или если что-то помешало запросу выполниться в принципе!

На сервер данные можно отправлять в формате JSON или через специфический объект FormData, это зависит от backend-разработчика:

**FormData –** объект создаётся при помощи new, он собирает все данные с форм, данные из всех инпутов формы передаются в FormData в формате:

[имя инпута]: [значение инпута]

И это довольно специфический объект, с которым мы можем взаимодействовать только определенными способами. Так было спроектировано при её разработке. В том числе, у FormData встроен метод forEach. А вот у обычных объектов его нет, поэтому код и не будет работать.

Важно всегда проверять наличие у инпутов форм свойства name=’name’, тк без него могут возникать ошибки!

Promise(ES6)

Данная технология помогает очень успешно и приятно работать с рызличными ассинхронными операциями. Ассинхронный код бывает, например, в таймерах, когда мы запускаем, что какая-то функция выполнится/будет повторяться через определённый промежуток времени; также при запросах на сервер, когда мы точно не знаем, через сколько ответит нам сервер.

**const req = new Promise(function(resolve, reject) {})**

Синтаксис: создаём новый объект Promise через new, во внутрь помещаем callback-функцию, как правило, с 2-мя аргументами – resolve, reject, которые означают функции, которые в будущем можно самим передавать. Resolve означает, что что-то выполнилось правильно, reject – не выполнилось.

У Promise есть огромное преимущество перед callback-функциями, мы можем возвращать Promise и then по цепочке. Когда одна ассинхронная операция выполнится, мы выоплним следующую, потом следующую и так по цепочке

Методы:

1. then() – выполняется на Promise в случае положительного исхода, т.е. это функция resolve(). Внутрь метода помещаем функцию, как аргумент для функции resolve внутри Promise
2. catch – нужен для обработки ошибок функции reject, которая является аргументом в Promise. Обычно прописывается после then()
3. finally() – позволяет выполнить действие в абсолютно любом исходе Promise, т.е. это действия, которые должны быть произведены абсолютно всегда! Обычно прописывается в самом конце
4. all([]) – во внутрь принимает массив с Промисами. Данный метод служит для того, чтобы мы точно убедились, что все наши Промисы уже выполнились
5. race([]) – во внутрь также передаётся массив с Промисами, но выполнится он тогда, когда один, какой-то из Промисов выполнился первым, нам не нужно ждать выполнения остальных, как в all([])

Что такое библиотеки. Библиотека axios.

**Библиотека** – это либо какой-то маленький участок кода, либо целый большой ресурс, который берёт и решает определённую проблему, но главное, что это уже готовое решение!

<https://cdnjs.com/> - cdn-сервера с библиотеками и фреймворками

**Библиотека axios –** это очень удобный ресурс, который позволяет работать с сервером: точно также получать данные, постить их и выполнять другие действия. Когда мы не хотим или нам просто некогда заниматься настройкой функций по получению или отправке каких-то данных, можно подключить данную библиотеку с готовым функционалом. Библиотека axios, в отличие от fetch, возвращает более развёрнутый ответ, а все данные лежат в свойстве data, но при этом обработка ошибок, автоматическая конвертация ответа в нужный тип и прочие плюшки идут без нашего участия и кода.

Как сохранить данные без БД. Работа с localStorage.

**localStorage** – это достаточно интересная технология, которая служит для хранения каких-то локальных данных, она позволит сохранять данные на сайтах без работы с сервером и БД, и вся эта информация сохранится, даже если перезагрузить страницу или закрыть браузер. По простому, **localStorage – это объект**, который встроен в браузер, и он может хранить различные данные. Если капнуть глубже, то это на самом деле свойство глобального объекта window, т.е. мы можем прописать:

window.localStorage

Этот объект уникален для каждого домена, это сделано в целях безопасности, ведь JS не может выходить за пределы вкладки, работать с другими вкладками или файлами на компьютере.

Объект localStorage можно использовать в огромном множестве случаев: человек устанавливает какие-то настройки сайта и при посторном заходе всё останется на местах, или установить какие-нибудь данные формы, или запомнить время, где пользователь остановился при просмотре видео и после продолжить с него, и т.д.

Сам объект localStorage НЕ резиновый и может содержать около 5мб информации, поэтому это не замена большим БД. И если вы его превысите — получите исключение QUOTA\_EXCEEDED\_ERR. Кстати, c его помощью можно проверять есть ли в вашем хранилище еще место.

Изображение выглядит как текст

Автоматически созданное описание

<https://learn.javascript.ru/try-catch> - **про конструкцию обработки ошибок try…catch…finally.**

Для того, чтобы работать с localStorage нужно знать 4 команды:

1. setItem(‘name key’, ‘value’) – служит для записи нового ключа, где первый аргумент – это название нашего ключа, а второй – значение ключа.

Если внутри localStorage уже есть такой ключ, то значение перезапишется!

1. getItem(‘name key’) – служит для получения данных из объекта localStorage, здесь только один аргумент – название ключа, который хотим получить
2. removeItem(‘name key’) – служит для удаления ключа, название которого помещаем как аргумент
3. clear() – служит для полной очистки нашего локального хранилища localStorage, прописываем без аргументов.

Изображение выглядит как текст

Автоматически созданное описание

В объекте localStorage можно хрнаить любые типы данные, не только примитивные, но и объекты, массивы, но в таком случае необходимо сделать серилизацию наших данных, иначе получим надпись Object. Самый простой вариант – перевести в формат JSON.

<http://dev-test.nemikor.com/web-storage/support-test/> - проверка вместимости локального хранилища

Регулярные выражения

**Регулярные выражения** позволяют очень удобно работать со строками: удалять, заменять части слов, искать какие-то кусочки строк, ограничивать ввод определённых знаков и т.д.

**Любое регулярное выражение состоит из 2х частей:**

1. Паттерн(шаблон) – это шаблон того, что мы ищем в этой строке или, что мы пытаемся удалить и т.п. Т.е. как-то мы ограничиваем определённый список символов или их порядок
2. Флаги. **Существует самых классических 3 флага**: i (если мы хотим найти что-то в независимости от регистра), g (когда пытаемся найти сразу несколько вхождений, а не одно), m (включает многострочный режим). Флаги можно комбинировать и прописовать подряд!

В JS для **создания регулярных выражений** можно использовать несколько синтаксисов:

1. При помощи конструктора new RegExp(‘pattern’, ‘flags’); но такой способ не особо популярен
2. /pattern/flags – это более простой и популярный способ. Если внутри /./ прописать точку, это значит, что мы берём абсолютно все элементы, которые попадут в строку! В таком случае, чтобы взять за символ отдельную точку, её нужно экранировать, т.е. поставить \. => /\./g

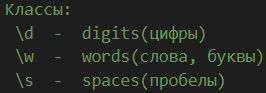
**Методы строк с регулярными выражениями:**

1. search() – пытается что-то найти в переданном аргументе, он всегда ищет только первое совпадение. Данный метод довольно простой, поэтому с флагом ‘g’ он не сработает!
2. match() – мощнее, чем предыдущий метод, т.к. результатом будет массив, свойства из которого можно потом как-то использовать. К тому же с этим методом можно использовать флаг глобальности ‘g’!
3. replace() – позволяет брать какой-то кусочек строки и заменять его на другое выражение. Внутри () прописываем 2 аргумента: что мы заменяем(можно прописать регулярное выражение) и на что мы изменяем

**Методы регулярных выражений:**

1. test() - проверяет строку на наличие совпадений с регулярным выражением. Метод возвращает true, если совпадения были найдены. И false, если нет.
2. D

В паттернах можно указывать строки, числа, но иногда мы хотим искать не что-то конкретное, мы хотим искать целый класс символов, например, цифры, слова, пробелы. **Для этого в регулярных выражениях есть такое понятие, как Классы!**



Существуют также обратные Классы, т.е. иногда нам нужно найти НЕ числа, НЕ буквы, НЕ пробелы, для этого используем похожий синтаксис, также экранируем, но используем большую букву!

Изображение выглядит как текст

Автоматически созданное описание

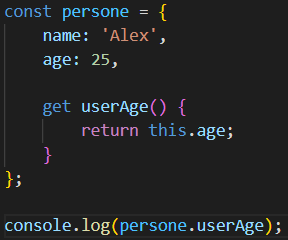
При использовании \W будет находиться любой символ в строке, кроме \w, то есть не буквы из латиницы, не знак подчёркивания и не цифра. В частности, русские буквы принадлежат этому классу.

<https://regex101.com/> - конструктор регулярных выражений

Геттеры и сеттеры(свойства объектов)

Вспомним, что у объект состоит из свойств и методов. Свойства – это то, что описывает наш объект, а методы – это то, что умеет делать наш объект. Свойства у объекта делятся на 2 типа – это свойства Данные(просто описывают наш объект) и свойства Акцессоры – это целый комплекс, который позволяет как присваивать, так и получать значения, но при этом во внешнем коде они будут выглядеть как обычные свойства объектов.

**Свойства акцессоры делятся на 2 большие группы: геттеры и сеттеры.** Геттеры получают значения, а сеттеры присваивают.

 => 25

Синтаксис: прописываем ключевое слово get и его название, далее синтаксис, как у обычного метода. Для вызова нам необходимо обратиться к данному геттеру по его названию, как к обычному свойству. Мы не ставим () и не вызываем геттеры, т.к. уже предполагается, что внутри какой-то функционал.

**В пару к геттеру можем установить и сеттер**, синтаксис будет похож и даже название, и код не будет ругаться на повторение, т.к. используется как пара get-set. Первый нам нужен для получения значения, второй – для установки значения.

Изображение выглядит как текст, экран

Автоматически созданное описание => 30

Инкапсуляция и приём «Модуль»

**Инкапсуляция** – один из принципов ООП – это отделение и сокрытие от внешнего мира внутренности программы, т.е. её переменных, функций и пр. В ООП это означает, что объект хранит своё состояние в приватном порядке и только методы объекта имеют доступ для его изменения. Это делается для защиты от вмешательства пользователя, также мы всегда без последствий можем улучшать и дорабатывать нашу программу, ну и это удобно, когда внутренности программы скрыта, а мы используем лишь её результат.

**Для скрытия внутренних деталий реализации скрипта используется такой приём «Модуль»!** Преимущества: во-первых, модули по своей струтуре сомодостаточны и независимы, поэтому код может быть легко изменён или вообще удалён, во-вторых, они обеспечивают чистоту глобального простанства (глобальных переменных должно быть как можно меньше) и помогают избежать проблему конфликтов с одинаковыми именами, ведь все данные Модуля существуют только в его собственной области видимости, в-третьих, инкапсуляция позволяет нам менять внутреннюю реализацию модулей безболезненно, поэтому мы сможем переиспользовать Модули каждый раз немного подстраиваясь под задачу и, возможно, улучшая Модуль.

**2 основных способа создания Модулей через нативную реализацию**:

1. Использование анонимной самовызывающейся функции (Immediately Invoked Function Expression — IIFE). Анонимной функция является, т.к. у неё нет имени, а самовызывающейся, т.к. после того, как она была объявлена, она сразу же вызывается на месте. Это очень интересный приём, который позволяет написать код, сразу же его запустить, и, соответвенно, получить локальную область видимости! Важно использовать внешние круглые скобки, т.к. если их не будет, то мы получаем Function Declaration, у которой должно быть имя, а не Function Expression!



1. Использование объектного интерфейса. Записываем наш Модуль в переменную и в ней уже возвращаем методы, доступные снаружи. Т.е. из анонимной самовызывающейся функции мы возвращаем объект, в который передаём ссылки на приватные перемнные и функции. Теперь мы можем использовать те методы, которые были экспортированы из объектного интерфейса.

По простому: наша анонимная самовызывающаяся функция создаёт объект, через который экспортирует только те методы и те свойства, которые действительно нужны снаружи!

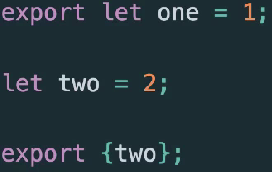
 => I am privat!

**Webpack** – сборщик модулей, работает с синтаксисом CommanJS (module.exports и require). Чтобы webpack мог собирать наш проект, необходимо установить его локально в наш проект и после запустить через команду npx webpack.

ES6 Modules

Новые стандарты нам также, как и синтаксис CommanJS позволяют формировать Модули при помощи спец. синтаксиса. Для того, чтобы что-то экспортировать мы можем написать ключевое слово export и далее уже указываем то, что мы экспортируем.

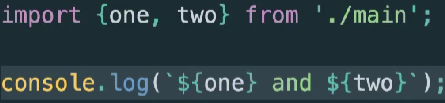
**Экспорт переменных / функций 2мя способами**:

 и Изображение выглядит как текст

Автоматически созданное описание

Самое главное в таких экспортах, чтобы у каждой сущности было своё имя, чтобы мы могли её экспортировать! Во втором случае обязательно заключать нашу сущность при экспорте в {} это называется именнованным синтаксисом.Можно экспортировать сразу несколько сущностей в одном файле!

Чтобы подключить(испортировать) значения в другой файл прописываем ключевое слово import, после ставим {} и прописываем имена наших сущностей, далее за скобками прописываем from и указываем путь к файлу! Здесь обязательно использовать {} для именнованного синтаксиса, т.к. экспортируется по факту большой объект, а сам синтаксис напоминает деструктуризацию.



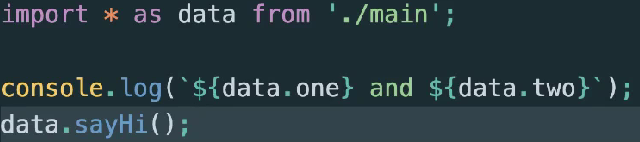
Но даже такой синтаксис необходимо собирать каким-то сборщиком Модулей, т.к. браузер НЕ умеет собирать всё в один рабочий скрипт! Можно также использовать Webpack.

**Доп.плюшки:**

1. Мы можем сразу при импорте переименовать наши сущности



1. Можно сразу импортировать все сущности, которые экспортируются из файла, для этого прописываем \* и даём названием через ‘as’. Далее обращаемся к сущностям, как к свойствам/методам объекта

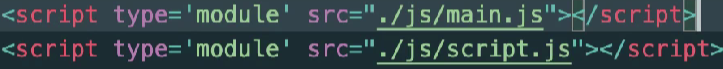
 => 1 and 2 => Hello

1. В Модулях помимо «именнованного синтаксиса» существует «экспорт по умолчанию», он может быть только один. При экспорте прописывается ключевое слово default и при импорте нет {}.

Изображение выглядит как текст

Автоматически созданное описание 

1. **Атрибут type=‘module**’ (используется редко), который можно применить к тегу script, чтобы он попытался использовать наши скрипты в качестве Модулей. Сначала у нас должны быть настроены экспорты и импорты, как описано выше, и далее это надо правильно подключить на нашу страницу. Первым подключаем файл с экспортными сущностями, а после уже файл с импортом. Также теперь придётся в наших файлах указывать полный путь с .js



НО это не значит, что браузер теперь сможет собрать всё это в один файл, он может только последовательно подключить эти файлы, используя импорт экспорт. Если таких модулей будет много, их все придётся подключать последовательно!

Обработка ошибок, "try..catch"

Обычно скрипт в случае ошибки «падает» (сразу же останавливается), с выводом ошибки в консоль. Но есть синтаксическая конструкция try..catch, которая позволяет «ловить» ошибки и вместо падения делать что-то более осмысленное.

<https://learn.javascript.ru/try-catch> - полное описание конструкции

Конструкция try..catch состоит из двух основных блоков: try, и затем catch, можно также указать блок finally, который выполнится при любом исходе. Работает она так:

1. Сначала выполняется код внутри блока try {...}.
2. Если в нём нет ошибок, то блок catch(err) игнорируется: выполнение доходит до конца try и потом далее, полностью пропуская catch.
3. Если же в нём возникает ошибка, то выполнение try прерывается, и поток управления переходит в начало catch(err). Переменная err (можно использовать любое имя) содержит объект ошибки с подробной информацией о произошедшем.

Создание своих ошибок (new Error)

Иногда нам необходимо формировать свои собственные блоки кода с Ошибками и предупреждать об этом разработчиков или пользователей. Для этого в JS есть специальный оператор ***throw***, который генерирует Ошибку, и как ошибку мы можем выбросить всё что угодно: строку, число или собственный объект. Но обычно для этого используют встроенный Класс Error.

 => Изображение выглядит как текст

Автоматически созданное описание

Объекты Ошибок также, как и в конструкции try…catch, содержат *name, message, stack*! На самом деле Error, который мы используем выше, это общий конструктор Ошибок, но есть и более специфические, которые будут отличаться уже готовым *name,* например, new SyntaxError, ReferenceError, TypeError и т.д, их использование может быть более удобным! В крайних случаях можно создавать и свои отдельные классы Ошибок, которые наследуются от встроенных!

Вывод Ошибки можно кастомизировать как захочется! Чем больше скрипт, тем больше возможностей для Ошибок и их видов, поэтому иногда оставляют возможность выбросить ошибки наружу(за блок catch) также через *throw*, если она вдруг серьёзная!

Превращаем код ES6+ в старый формат ES5. Babel, Core.js и полифиллы

Чтобы наш продукт отлично работал и правильно отображался и на старых версиях браузера, нам необходимо использовать доп. плюшки:

1. **Babel** – это трансплитер, т.е. инструмент, который берёт код в новом формате и переводит его в старый формат, который понятен более старым браузерам. Способы использования: напрямую каждый файл прогонять через эту программу, или подключить эту утилиту через npm к сборщикам модулей/проектов (Webpack) <https://babeljs.io/>
2. Полифиллы – это определённые участки кода, которые эмулируют поведение современных стандартов. Существует **библиотека Core.js**, которая включает в себя огромное кол-во полифиллов, в ней есть встроенная функция, которая позволяет проверить код и определить только те полифиллы, которые действительно нужны, в отличии от babel/polyfill.
3. <https://browserslist.dev/?q=bGFzdCAyIHZlcnNpb25z> и <https://browsersl.ist/> - проверка браузеров

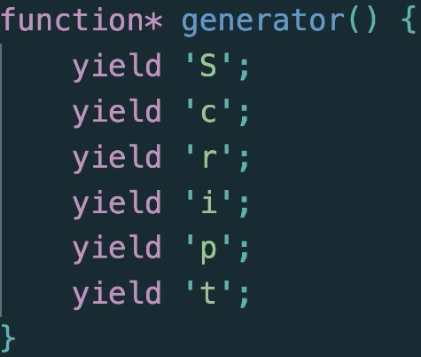
Бывает, что даже при использовании библиотеки Core.js, полифиллы от Babel, какой-то из полифиллов не будет работать. В таком случае это будет видно при тестировании, что что-то не работает. В таком случае нам необходимо установить эти недостающие полифиллы вручную. Для этого просто воспользуемся поиском в браузере.

Когда мы какие-то пакеты устанавливаем во внутрь нашего npm-проекта, они заносятся в папку node\_modules, и т.к. внутри этой папки лежат все эти скрипты, библиотеки и т.д., то мы можем просто брать их оттуда и через систему импортов-экспортов ES6 импортировать их во внутрь наших скриптов. Синтаксис подключения (импорта) npm-пакетов будет отличаться от подключения файлов: прописываем import и в кавычкая БЕЗ точки указываем название пакета. Также можно подлкючать, например, slick-slider. Преимущество в том, что теперь получится всего один JS файл, который будет содержать в себе все зависимости, а не куча скриптов, которые будут подключены в конце страницы!

Функции-генераторы

Данная функция генерирует нам какой-то результат и делает она это последовательно. Чтобы **создать функцию-генератор**, нам необходимо прописать **function\* name(){}** или function \*name(){}, но лучше пользоваться первым вариантом!

Т.к. функция-генератор даёт нам постоянно какой-то новый результат, то должны внутри функции прописать ключевое слово **yield** именно оно отвечает за этот функционал.



При первом вызове нам вернётся ‘S’, при втором ‘c’ и т.д. Но, чтобы вызвать следующий шаг нашего генератора, необходимо использовать встроенный метод **next()** Когда функция срабатывает, возвращается объект, в котором у нас есть 2 поля: value, где получаем значение и done, которое означает выполнился наш генератор полностью или нет (станет true, когда наша функция уже вернула последний шаг и вызывается ещё раз, тогда value станет undefined).

Изображение выглядит как текст

Автоматически созданное описание => Изображение выглядит как текст, знак, закрыть

Автоматически созданное описание

Чтобы **сразу получить значение** внутри value, нам достаточно обратиться к нему, после метода next(), через точку, т.к. нам возвращается Объект!

Чтобы нам каждый раз не создавать вручную новый результат, мы можем делать это автоматически, например, через цикл for. A cаму функцию-генератор можно перебрать через перебирающую конструкцию for of, вместо того, чтобы всё время вызывать метод next()!

JS анимации, requestAnimationFrame

Самый простой способ создания анимаций – CSS3 анимации, но у них есть некоторые ограничения: во-первых, CSS не умеет нормально анимировать Canvas-элементы, во-вторых, не получится создать какую-то «рваную» нестандартную анимацию. Чтобы обойти такие ограничения, лучше использовать JS анимации, их существует как минимум 2 вида:

1. Через setInterval/setTimeout, но это не совсем оптимизированный и хороший способ создания анимаций, тем более сложных, т.к. он нагружает ресурсы нашего компьютера
2. **requestAnimationFrame()** – принимает в себя только один аргумент – функция-анимации, которая нас интересует. Позволяет нам запускать какие-то функции(в виде callback) в качестве анимаций, при этом эта функция запускается так, что она берет анимацию и подстраивает её под частоту обновления браузера, таким образом, анимация будет происходить в тот момент, когда идёт обновление страницы, и мы точно также будет видеть плавную анимацию, но при этом нагрузка на брузер значительно снижается, в отличие от анимаций через setInterval/setTimeout! Также если на странице есть другие анимации, то requestAnimationFrame позволит все анимации выполнять одновременно.

Отменять анимации можно также, как происходит удаление таймеров, т.е. мы помещаем вызов нашей анимации в переменную, которая будет содержать уникальный идентефикатор. **Чтобы отменить анимацию используем cancelAnimationFrame(id).**

Web Animation API

Данная технология позволяет удобно создавать анимации на JS по типу CSS3 анимации и управлять ими. Для создания анимаций нам понадобится специальный **конструктор Animation**, его можно создать 2мя способами:

1. Более сложный – через конструктор new Animation(), в который передаём аргументы: effect – найстройки анимаций и timeline
2. Использовать **метод animate(keyframes, options)** на элементе страницы. В таком случае анимация сразу применяется к элементу, а затем проигрывает её. Во внутрь () можно передать настройки ключевых кадров. Первым аргументом передаём массив, это должен быть массив объектов с ключевыми кадрами: начальная точка, промежуточные точки и конечная. Под ключевыми кадрами подразумеваются CSS свойства, который мы будем менять (как они выглядели в начале и как будут выглядеть в конце). Вторым аргументом передаём опции, которые задаём для нашей анимации, в виде объекта с настройками.

Event loop, подробная работа ассинхронных и синхронных операций

Ассинхронными операциями являются setTimeout/setInterval, запросы на сервер, а если капнуть глубже, то и все callback-функции.

Event loop или событийный цикл работает с

<https://clck.ru/RHJoa> - можно посмотреть как работает Event Loop

Вопросы и задачи собеседований:

* **Почему в консоли браузера при выполнении кода мы когда-то можем видеть undefined?** – console.log() это функция (console – большой объект, а log – это его метод, т.е. функция, записанная внутри объекта), а функция, как известно, всегда что-то возвращает и без конкретно заданного return функция будет возвращать undefined.
* **Какое будет выведено значение: let x = 5; alert( x++ ); ? =**> 5 (важно помнить про инкремент и декремент)

**Изображение выглядит как стол

Автоматически созданное описание**

* **Чему равно такое выражение: [ ] + false - null + true ? =**> NaN (первое действие вернёт false, а далее мы будем пытаться произвести не математическое вычитание и нам вернётся NaN). P.S. пустой массив превращается в пустую строку и при сложении возвращает String!
* **Что выведет этот код: let y = 1; let x = y = 2; alert(x); ? =**> 2 (т.к. это примитивы, то передача данных по значению)
* **Чему равна сумма [ ] + 1 + 2? =**> 12 (пустой массив при сложении с 1 возвращает строку «1» и при сложении с числом 2 вернёт «12»)
* **Что выведет этот код: alert(“1”[0])?** => 1 (т.к. нулевой индекс в данной строке это и есть 1)
* **Чему равно 2 && 1 && null && 0 && undefined? =**> null (т.к. логический оператор «И» запинается на лжи, а null это первое значение в данной строке, которое вернёт false)
* **Есть ли разница между выражениями? !!( a && b ) и (a && b)? =**> Да (т.к. первое выражение вернёт Boolean)
* **Что выведет этот код: alert( null || 2 && 3 || 4 ); ? =**> 3 (приоритет выполнения у оператора “И”, и т.к. 2 и 3 это true, то возвращается последнее «правдивое» значение 3. Далее идут операторы «ИЛИ», которые запинаются на правде, и после первого сравнения останется 3, а дальше код уже не пойдёт)
* **a = [1, 2, 3]; b = [1, 2, 3]; Правда ли что a == b ? =**> Нет (т.к. это 2 абсолютно разных массива хоть и с одинаковыми значениями)
* **Что выведет этот код: alert( +«Infinity» ); ? =**> Infinity (выведет всю ту же строку, однако тип данных станет Number из-за унарного +)
* **Верно ли сравнение: “Ёжик” > “яблоко” ? =**> Нет (особенности uniCode)
* **Чему равно 0 || && || 2 || undefined || true || falsе ? =**> 2 (“ИЛИ” запинается на правде!)