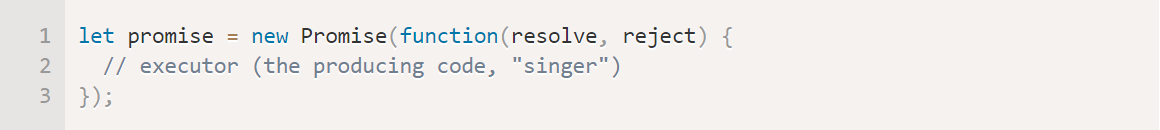
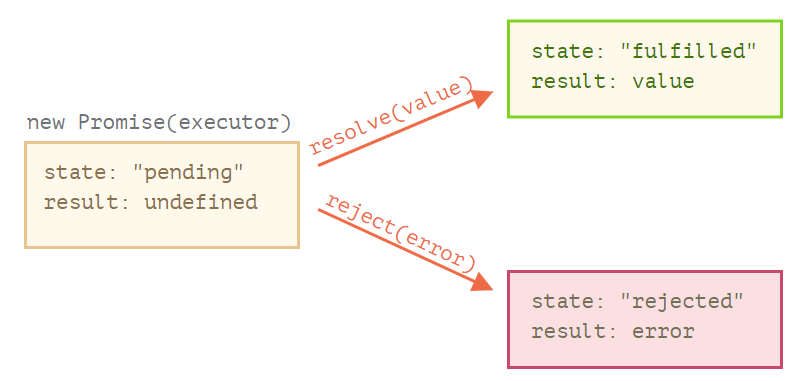
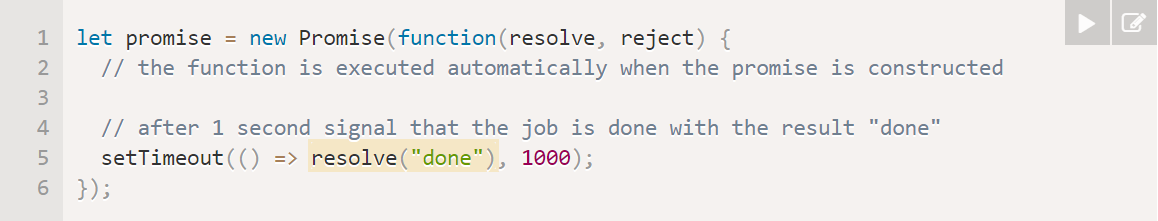
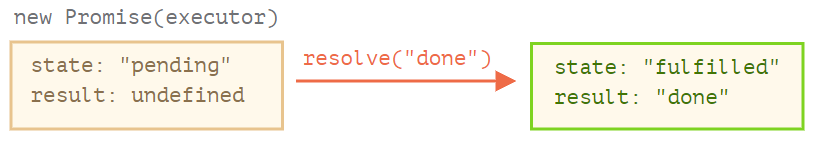
# **03.08.2020 5 часов Отчет: начал изучать promises 1-4**

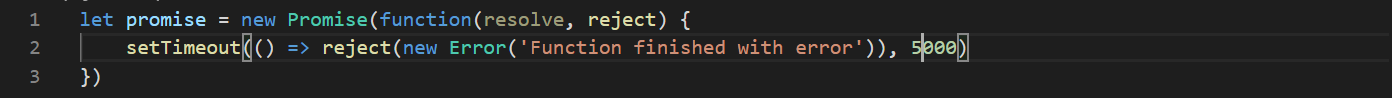
**Promises**Синтаксис создания:   
Функция, которая передается в promise, называется executor. Когда promisе создался, исполнитель запускается автоматически. Resolve и reject – колбэки, которые предоставляет promise

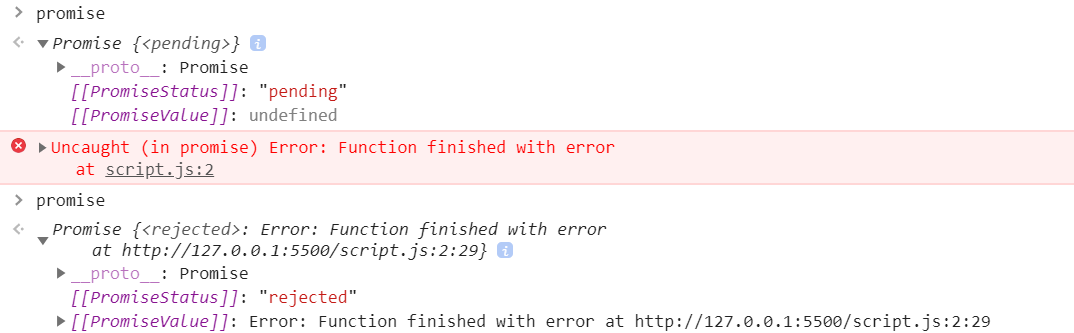
Resolve(value) вызывается, если работа колбэка завершилась успешно с результатом value  
Reject(error) вызывается, если есть ошибки, error – данные об ошибке

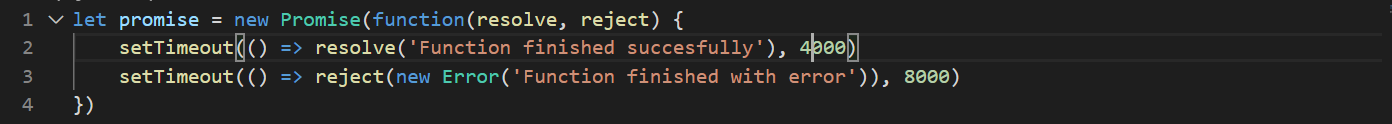
У объекта promise, возвращаемого конструктором new Promise, есть внтуренние свойства:  


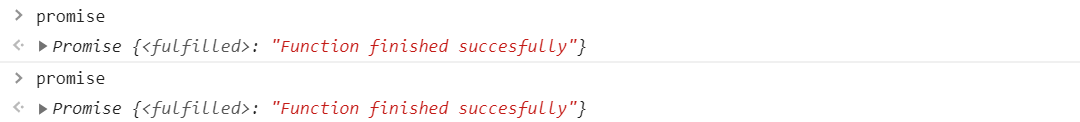
В примере ниже, мы создаем promise, причем, запустив код, можем наблюдать две вещи:  
1) Функция-исполнитель запускается сразу же при вызове new Promise  
2) Исполнитель получает две встроенные функции resolve и reject

В результате promise поменяет свое состояние на следующее: 

Теперь запустим код ниже, и вызовем ошибку специально:  


Замечаем, что promiseStatus и promiseValue сменились на rejected и Error соответственно 

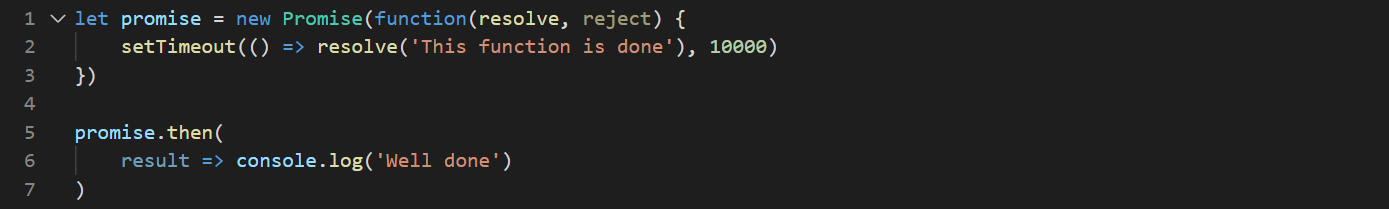
Также хотелось бы отметить, что executor может вернуть только resolve или только reject: 

Посмотрев в консоли состояние promise в разное время замечаем, что executor вернул resolve  


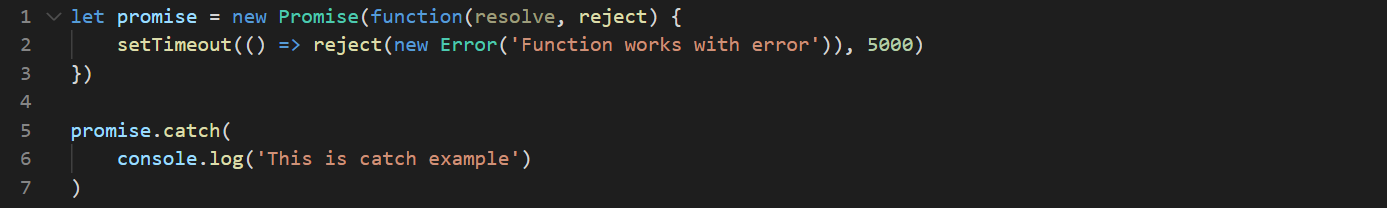
Свойства state и result – это внутренние свойства объекта promise, к нему мы не имеем прямого доступа. Для обработки результата следует использовать методы .then .catch .finaly

Объект promise – связующее звено между исполнителем и функциями-потребителями, которые получают либо результат, либо ошибку

.then – является наиболее важным методом, может принимать в себя два параметра:  
1) функция, которая выполняется, когда промис переходит в состояние «выполнен успешно», и получает результат  
2) функция, которая выполняется, когда промис перходит в состояние «выполнен с ошибкой», и получает ошибку

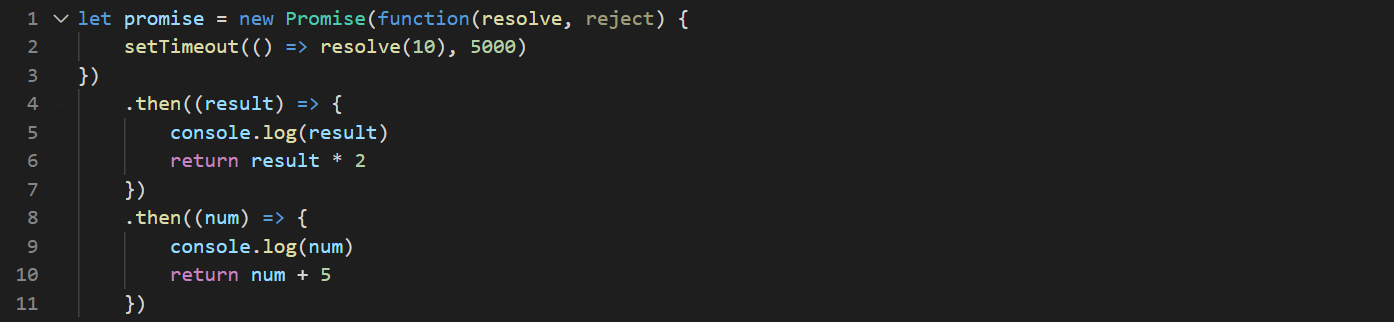
Рассмотрим первый случай, когда промис перешел в состояние «выполнен успешно»   
Что делает данный код?  
1) Изначально promise создается с помощью конструктора new Promise и имеет status: pending, value: undefined  
2) Через 10 секунд вызывается функция resolve, которая меняет свойства promise на status: fulfilled, value: This function is done  
3) Т.к promise поменял свой статус на fulfilled, то в следующем методе then сработает console.log со значением Well done

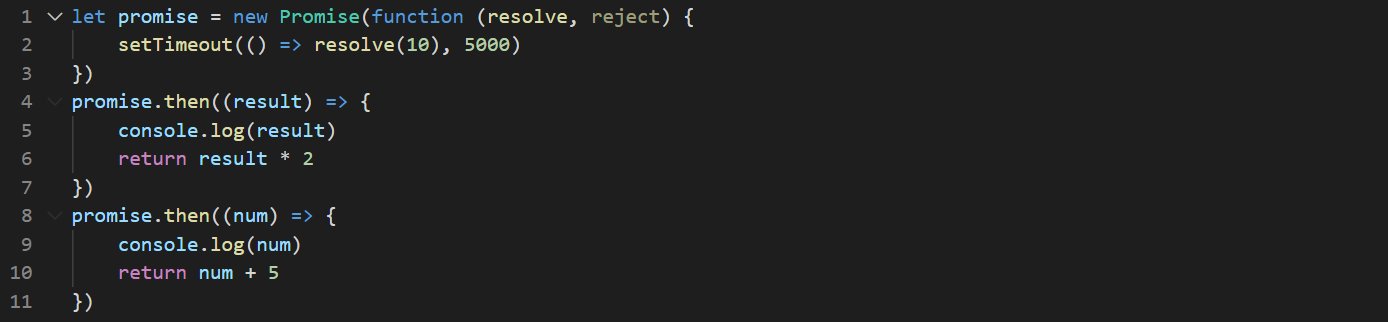
Если нам нужно, чтобы метод .then реагировал также на ошибку, то после 6ой строчки можно вставить следующий код: error => console.log(error)

Если мы хотим обработать ошибку, то можно использовать метод .catch Или то же самое можно записать с помощью then, передав первый параметр, как null

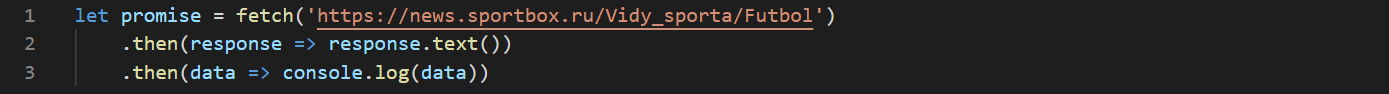
.finally – метод, который выполнится в любом случае, в не зависимости от того, как завершил работу promise

Особенности метода .finally:  
1) .finally не имеет аргументов т.к мы не знаем, как был завершен promise  
2) .finally не обязательно может быть последним методом в цепочке promise  
3) .finally = .then(f, f), но не нужно дублировать f

**Цепочка promise**Цепочка, созданная ниже является правильной  

А эта является неправильной. Значения в ней будут обрабатываться всегда относительно числа 10  
 



**Fetch**Во frontend разработке promise очень часто используют для того, чтобы делать запросы по сети. В примере ниже мы будем использовать метод fetch для того, чтобы сделать запрос на сервер и получить информацию   
Этот код запрашивает по сети url и возвращает promise. Если promise отрабатывает успешно, то он возвращает объект response, который присылает заголовки ответа  
Для того, чтобы прочитать полный ответ, нужно использовать метод response.text()

**Промисы: обработка ошибок**Цепочки промисов хорошо подходят для перехвата ошибок. Самый легкий способ перехватить все ошибки – добавить .catch в конец цепочки

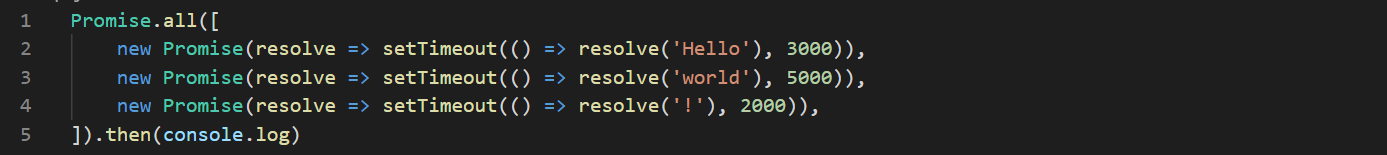
Что будет если ошибка не будет обработана? То есть если будет ошибка, но не будет указано метода .finally или .catch который обработал бы ошибку?  
В таких случаях движок JS отслеживает такие ситуации и генерирует глобальную ошибку

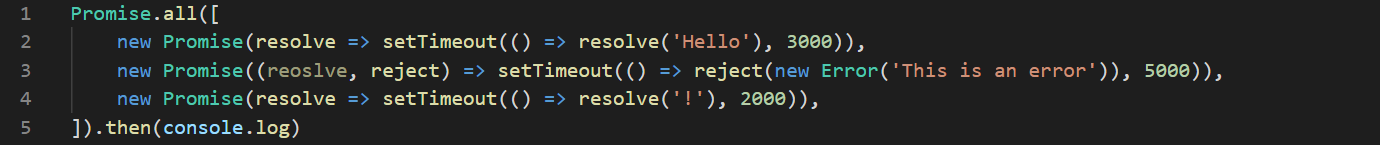
В браузере можно поймать ошибки, используя событие unhandlerejection  
Если происходит ошибка, и отсутствует ее обработчик, то генерируется событие unhadlerejection и соответствующий объект event, который содержит информацию об ошибке

# **04.08.2020 5 часов Отчет: закончил главу Promise, почти прошел Module**

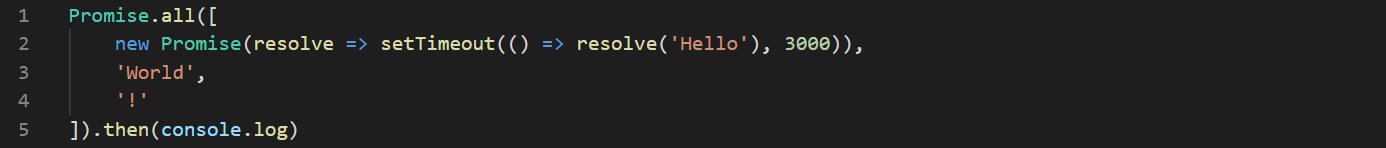
**Promise API**В классе Promise есть 5 статических методов

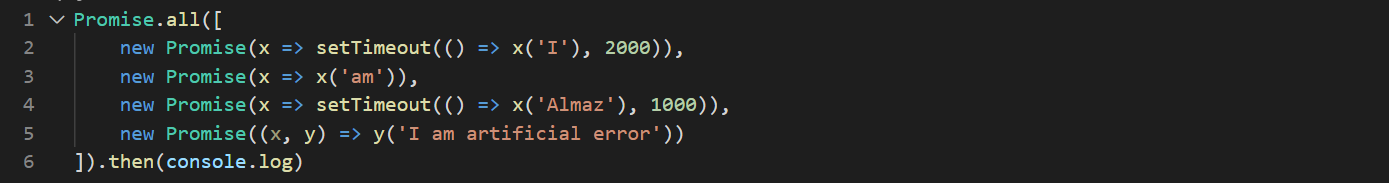
**Promise.all** – используется, когда нужно запустить множество promise параллельно и дождаться, пока они все выполнятся

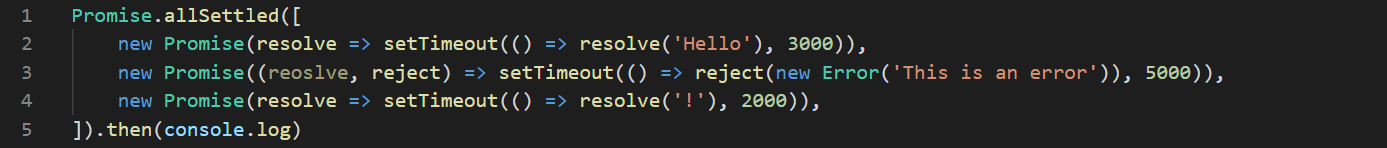
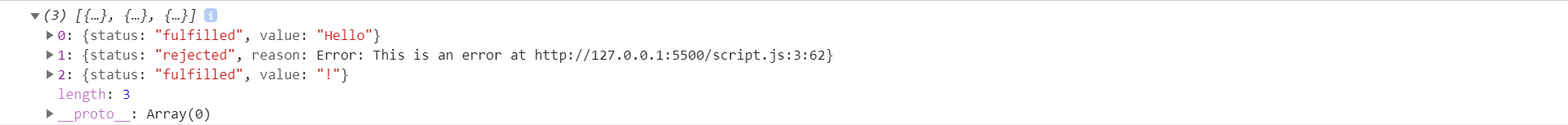
Метод Promise.all принимает массив промисов и возвращает новый массив  
Новый промис завершится, когда завершится весь переданный список промисов, и его результатом будет массив их результатов   

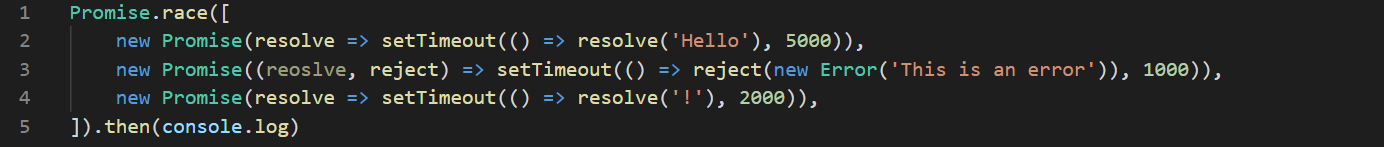

Если хотя бы один промис завершается с ошибкой, то Promise.all возвращает эту ошибку  

Если один промис завершается с ошибкой, то весь Promise.all завершается с ней, полностью забывая про остальные промисы в списке. Их результаты игнорируются

Обычно Promise.all принимает массив из объектов Promise. Но если любой из этих элементов массива не является промисом, то тогда он передается в массив как есть  
 

В примере ниже мы рассмотрели возможность писать без resolve и reject, заменив их на x и y

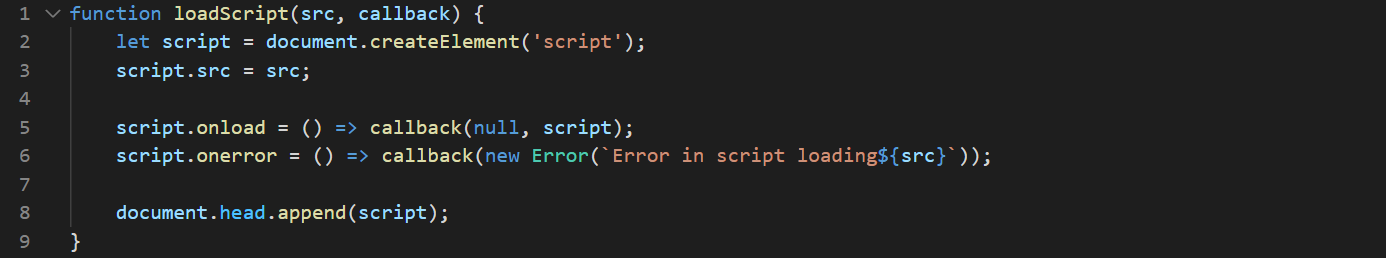
**Promise.allSettled** в отличие от Promise.all не будет прекращать свою работу в случае обнаружения ошибки в объекте Promise. Promise.allSettled полностью обработает все промисы, вернет массив объектов с ключами status, value  
 

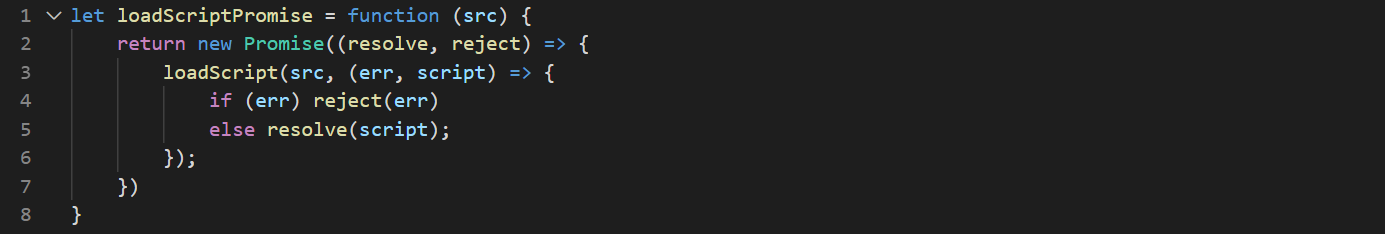
**Promise.race** – возвращает результат промиса, который завершил работу самый первый   
В данном примере Promise.race вернет ошибку, так как этот промис завершится быстрее всех  
 

**Promise.resolve(value) –** создает успешно выполненный промис с результатом value  
  


Promise.reject(error) – создает промис, завершенный с ошибкой error  
  


**Промиссификация –** функция, которая принимает колбэк, и возвращает промис  
Такие преобразования часто необходимы, так как многие функции и библиотеки основаны на колбэках, а использования промисов более удобно

Промиссифицируем эту функцию

В результате получим:  


**Микрозадачи**Обработчики промисов .then .catch .finally всегда асинхронны

Рассмотрим данный код  
 

Казалось бы, что порядок должен быть иным, однако из-за «очереди микрозадач» код ведет себя иначе

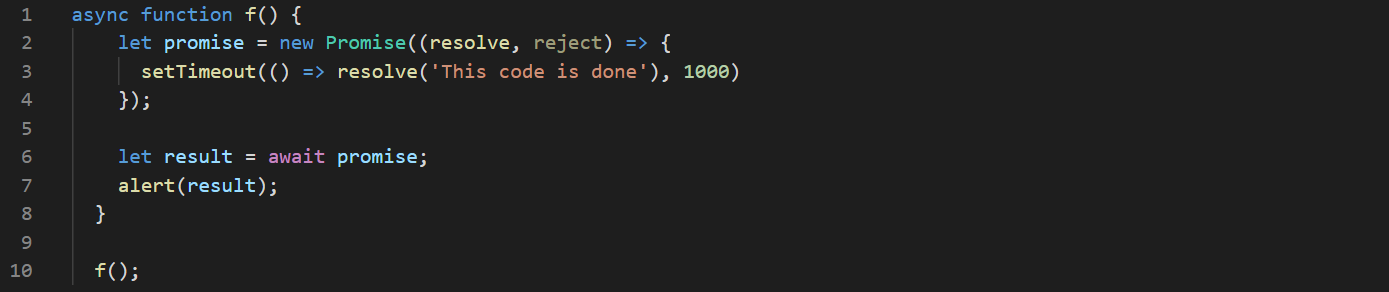
В спецификации сказано:  
- очередь выполняется по принципу «первый пришел – первый ушел». То есть задачи, попавшие первыми, выполняются тоже первыми  
- выполнение задачи происходит только в том случае, если больше ничего не запущено. Или, проще говоря, когда промис выполнен, его обработчики .then .catch .finally попадают в очередь

Для того, чтобы добиться «правильного» порядка, необходимо написать следующий код:  
  

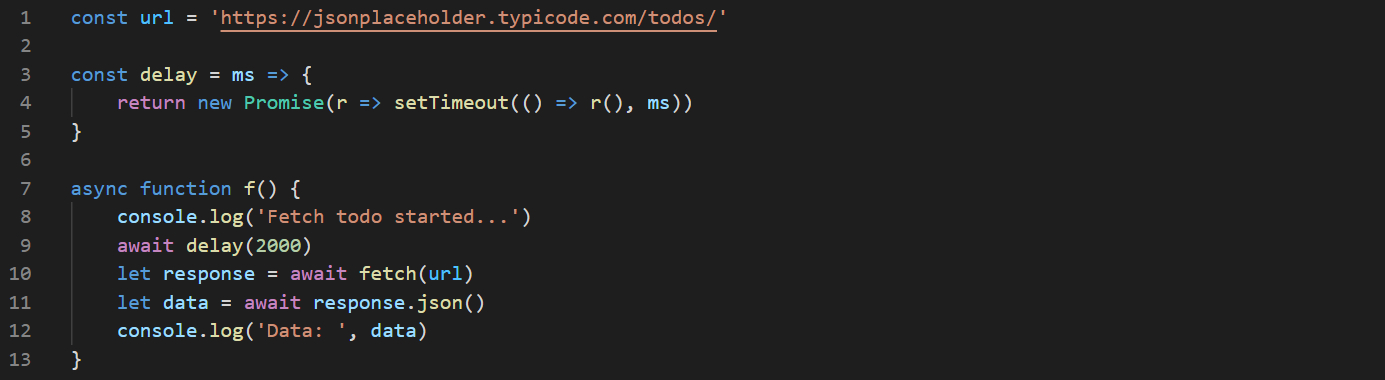

**Async/await  
Async –** ключевое слово, которое ставится перед функцией. У слова async один простой смысл – эта фукнция всегда возвращает промис.

Функция ниже вернет промис, который завершился с результатом «Hello world»

Убедимся в этом, посмотрев значение f в консоли  


**Await** – ключевое слово, которое работает только внутри async-функций. Это ключевое слово заставит интерпретатор JS ждать до тех пор, пока промис справа от await не выполнится. После чего оно вернет его результат, продолжится выполнение кода 

Рассмотрим код ниже   


Теперь добьемся такого же результата с помощью async и await  


Получив такой же результат, как в примере выше, убеждаемся что разницы никакой нет. Async и await – синтаксический сахар

**Модули**Когда мы работаем с крупными проектами, то держать весь код в одном файле неудобно. Сейчас разработчики стремятся к тому, чтобы «разбить» такие крупные файлы на множество. Модуль обычно содержит класс или библиотеку

Долгое время в JS не было модулей, потому что с момента появления JS скрипты были простыми и соответсвенно не было необходимости разбивать их на файлы

Система модулей появилась в JS в 2015 году. На данный момент она поддерживается многими браузерами и Node.js

Модуль – это файл. Один скрипт – один модуль

Модули могут загружать друг друга. Они используют ключевые слова как import и export  
export нужен для того, чтобы дать согласие на вынос модуля  
import нужен для того, чтобы вставить модуль в модуль

Здесь мы создаем функцию, которую готовы экспортировать: 

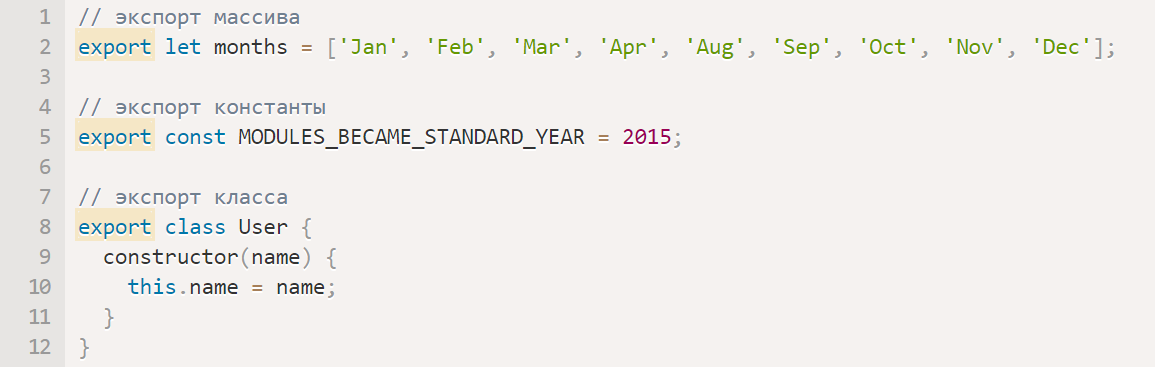
А здесь мы импортируем функцию sayHi с модуля sayHi.js

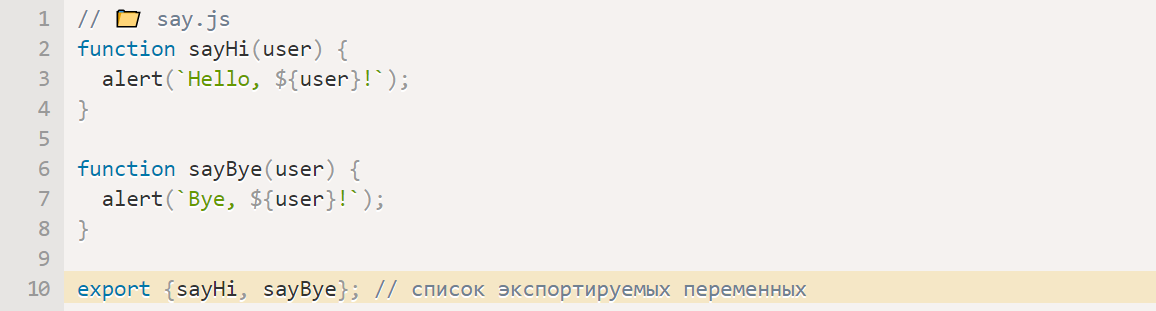
Чтобы пользоваться модулями, достаточно при объявлении скрипта указать type=«module»  

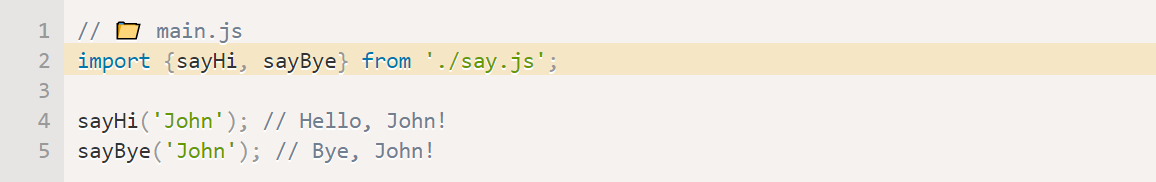

Отличия модулей от обычных скриптов:  
1) В модулях всегда используется режим use-strict  
2) Модули имеют свою область видимости. Т.е переменные и функции, объявленные внутри модулей не доступны для скриптов. Если нам нужно сделать глобальную переменную для всего документа, то мы ее можем объявить с помощью window (пример: window.name)  
3) Код в модуле выполняется только один раз. Даже если импортировать модуль в разные файлы, то код выполнится только в одном файле  
4) При импорте объектов в несколько файлов изменения сделанные в одном файле будут отражены в других  
5) this не определен в модуле. Т.е this===undefined

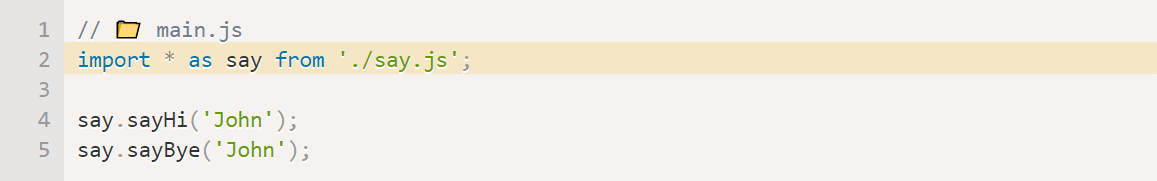
Особенности в браузерах:  
1) Модули всегда выполняются в режиме defer, т.е   
- загрузка внешних модулей не блокирует html  
- модули всегда ожидают полной загрузки html, только потом выполняются  
- сохраняется относительный порядок скриптов  
2) Атрибут async работает во встроенных скриптах, т.е для немодульных скриптов атрибут async работает только на внешних скриптах. Скрипты с ним запускаются сразу по готовности, они не ждут другие скрипты или html  
3) Два внешних скрипта имеющих тип module и одинаковый src запустят только один скрипт  
4) Не допускаются «голые» модули. Нужно указывать путь правильно (пр: ./Index.js)  
5) Старые браузеры не понимают type=«module», для подстраховки можно писать nomodule

Как работают инструменты сборки (пр: Webpack)  
1) Сборщик берет «основной» модуль, который мы собираемся поместить в script type=module  
2) Анализирует зависимости (импорты, испорты импортов и тд)  
3) Собирает один или несколько файлов (можно настроить) со всеми модулями  
4) В процессе сборки могут происходить и другие процессы:  
- недостижимый код удаляется, - полученный файл можно минимизировать  
- неиспользуемые экспорты удаляются  
- удаляются операторы для разработки (пр: console, debugger и тд)  
- новый синтаксис JS можно трансформировать в предыдущий стандарт с помощью Babel  
**Импорт и экспорт**  
Мы можем импортировать и экспортировать абсолютно все: массив, переменную, объект и тд

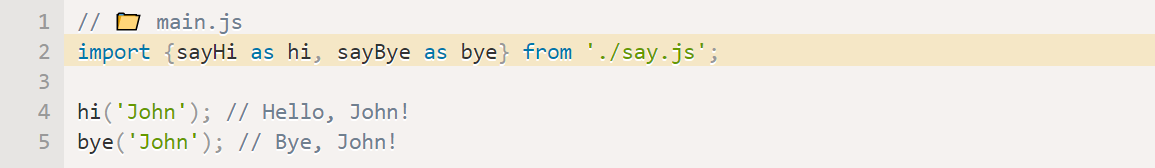
Экспортировать можно до объявления:  


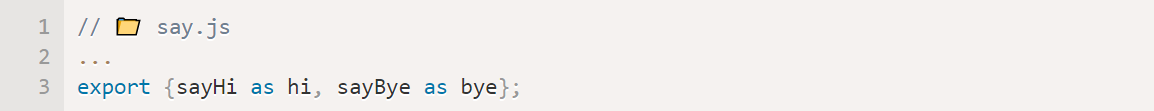
И отдельно от объявления: 

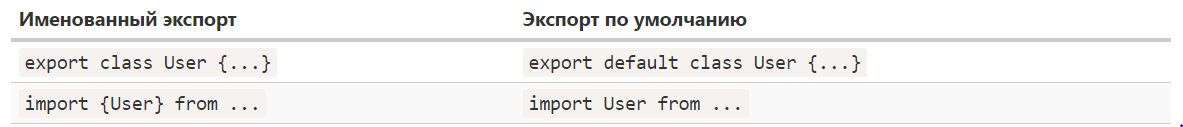
Для того, чтобы импортировать конкретные файлы, нам следует их указать при импорте:  


Или можно импортировать все с этого файла с помощью \*:  


Несмотря на то, что импортировать все кажется более удобным способом, есть причины по которым делать этого не рекомендуется:  
1) Если мы сделали 5 импортов, прописали их «вручную», но используем только один импорт. То в таком случае Webpack удалит другие 4. Но если мы сделали бы это через \* - то Webpack не удалит лишние импорты  
2) Импортируя «вручную» мы получаем короткие названия (пр: *sayHello* против *say.sayHello*)  
3) Явное перечесление импортов делает код более понятным. Это упрощает поддержку и рефакторинг кода

Также при импорте мы можем указывать другие имена  


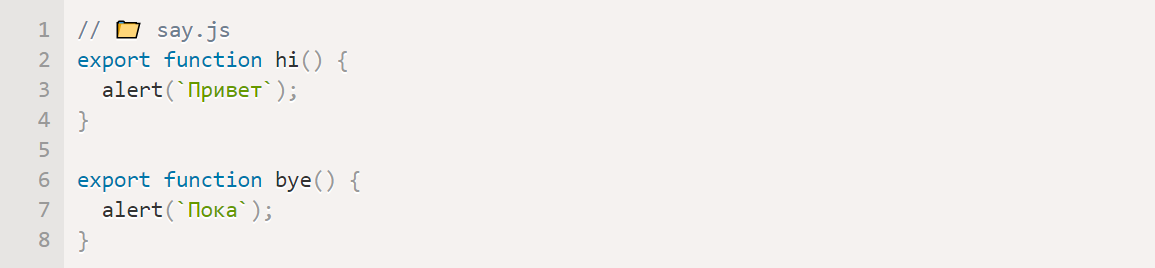
Таким же образом мы их можем и экспортировать  


**Export default**

# **05.08.2020 5 часов Отчет: закончил изучать Модули, перешел на Документ, прошел атрибуты и свойства**

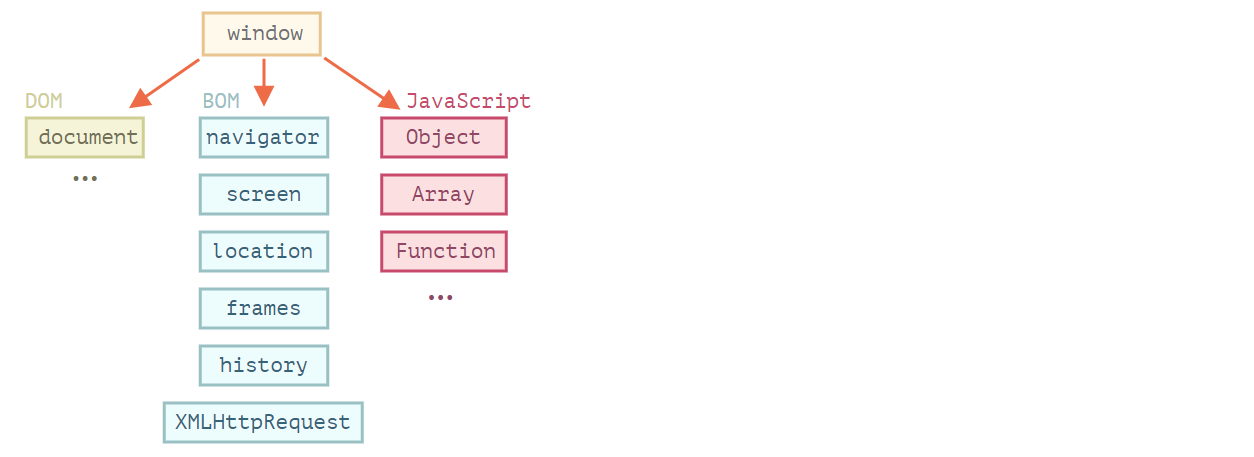
**Динамические импорты**Те инструкции, которые мы рассматривали в предыдущих главах, называются статическими. Они имеют достаточно простой и строгий синтаксис:  
1) Путь к импорту должен быть строковым примитивом и не может быть вызовом функции  
2) Нельзя делать импорт в зависимости от условий

**Выражение import() –** загружает модуль и возвращает промис, результатом которого становится объект модуля, содержащий все импорты

Если у нас есть такой модуль say.js

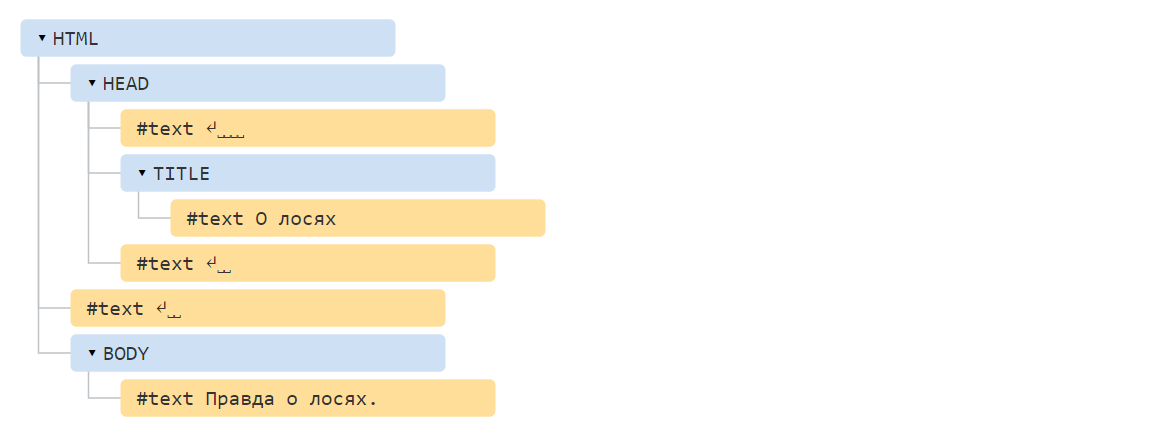
То динамический импорт может выглядеть вот так  

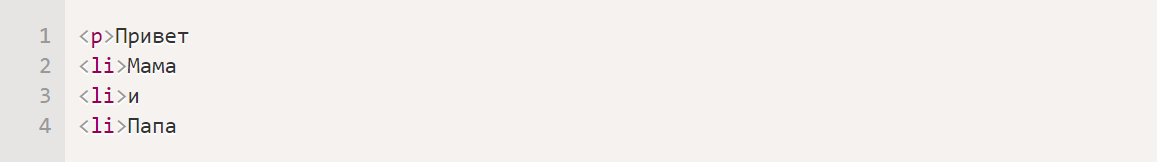

Динамический импорт работает в обычных скриптах и не требует указания type=module

**Браузерное окружение, спецификации**

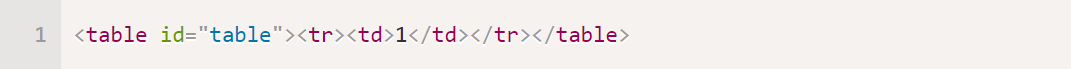
Стандарты:  
DOM – описывает структуру документа, манипуляции с контентом и события  
CSSOM – описывает файлы стилей  
HTML – описывает HTML и BOM

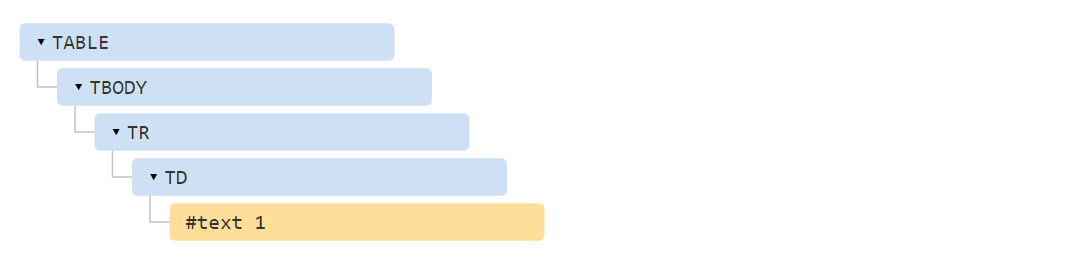
**DOM-дерево**В соответствии с DOM каждый HTML-тег является объектом, как и текст который находится внутри

В примере ниже видим, как работает дерево, причем оно учитывает пробелы и переносы строки

При генерации DOM браузер сам обрабатывает ошибки в документе, закрывает теги и так далее  
Есть такой документ с незакрытыми тегамиНо DOM будет нормальным, потому что браузер сам закроет теги и восстановит детали

Важный «особый случай» – работа с таблицами. По стандарту DOM у них должен быть <tbody>, но в HTML их можно написать (официально) без него. В этом случае браузер добавляет <tbody> в DOM самостоятельно

Для такого кода  


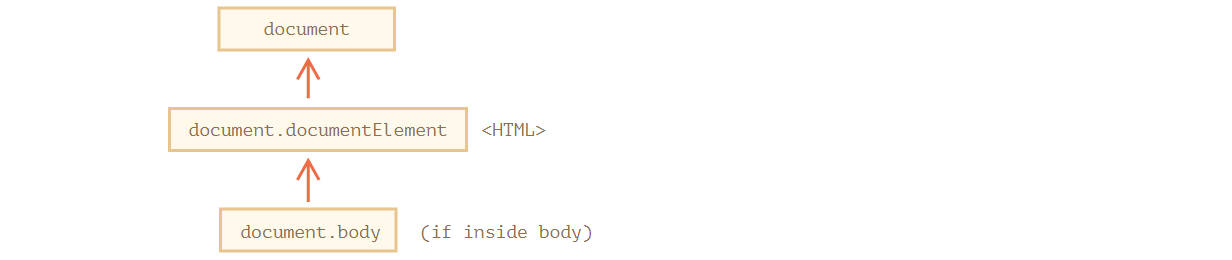
DOM-структура будет такой  


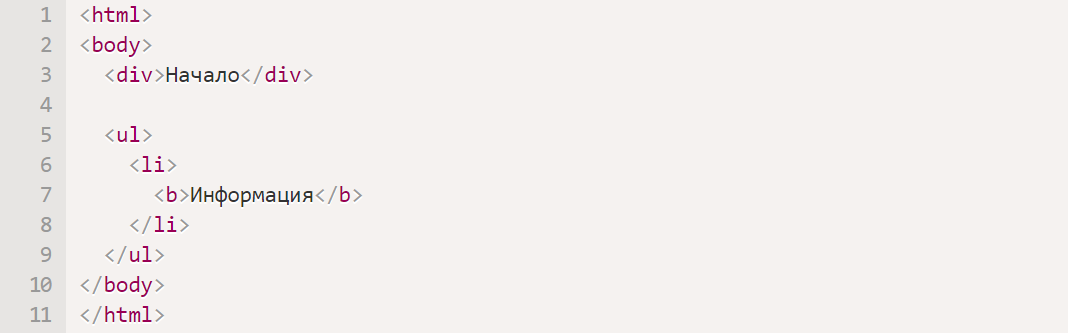
Существуют и другие типы узлов, кроме элементов и текстовых узлов, например комментарий  

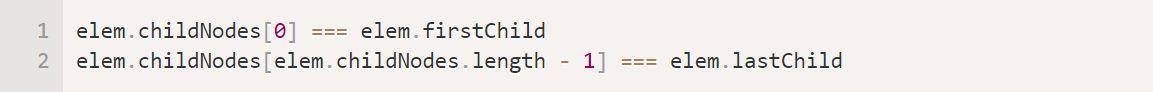

Существует важное правило: если есть что-то в HTML, то оно должно быть и в DOM

Существует 12 типов узлов в DOM, но мы будем работать с 4:  
- document  
- html-теги  
- текстовые узлы  
- комментарии

Для того, чтобы проанализировать DOM, можно зайти в режим разработчика, в консоль

**Навигация по DOM-элементам**

В примере ниже div, ul – дети для body, а div, ul, li, b – потомки для body

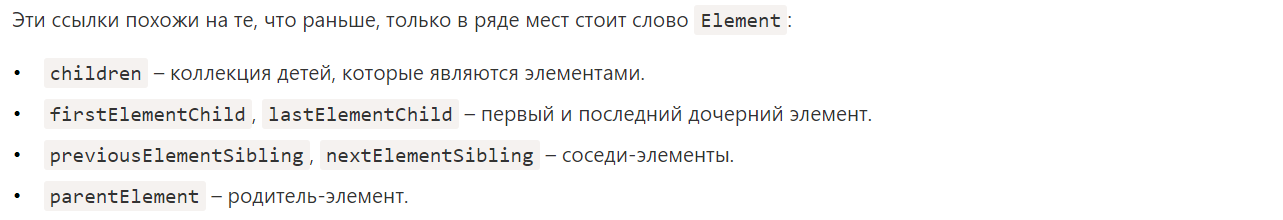
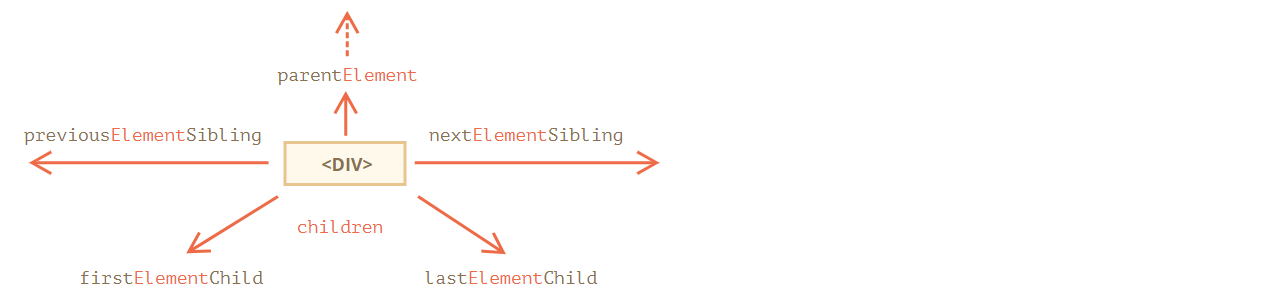
**DOM-коллекции**

childNodes – не массив, а коллекция. Поэтому мы можем ее перебирать с помощью for of, но не можем использовать методы массивов. Однако если мы хотим использовать именно методымассивов, но можем это сделать с помощью Array.from:



DOM-коллекции – только для чтения. Мы не можем заменить один узел на другой написав childNodes[i]=… Для изменения DOM требуются другие методы

**Соседи и родитель**Соседи – это узлы, у которых один и тот же родитель. 



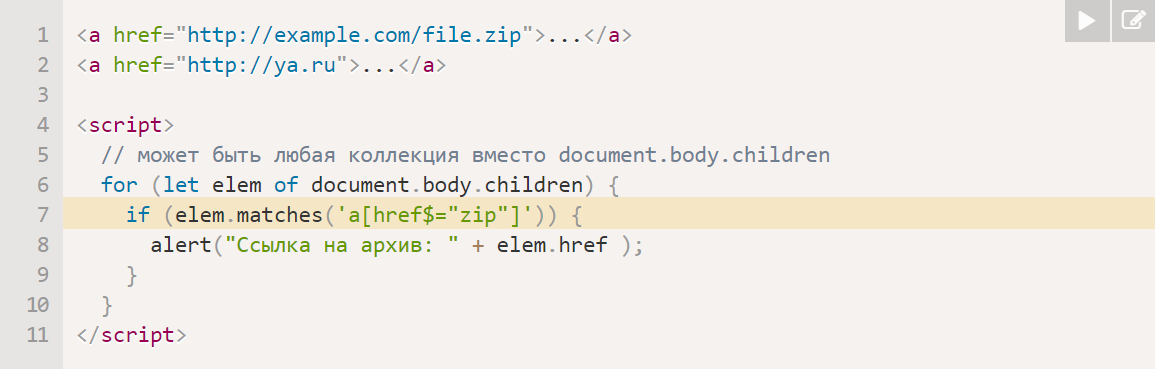
**Поиск по документу  
.getElementById**Значение id должно быть уникальным. Если в документе есть несколько элементов с одинаковым id, то document.getElementById будет непредсказуемым, он будет возвращать любой из этих id случайным образом

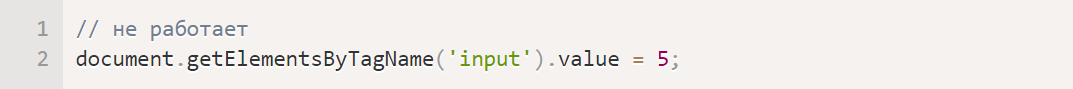
Метод .getElementById можно вызвать только для объекта document, он осуществляет поиск по id внутри документа. Нельзя использовать anyElement.getElementById

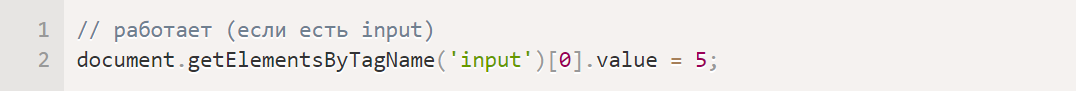
.**querySelectorAll(css) –** самый универсальный метод поиска, который возвращает объект, удовлетворяющий css селектору

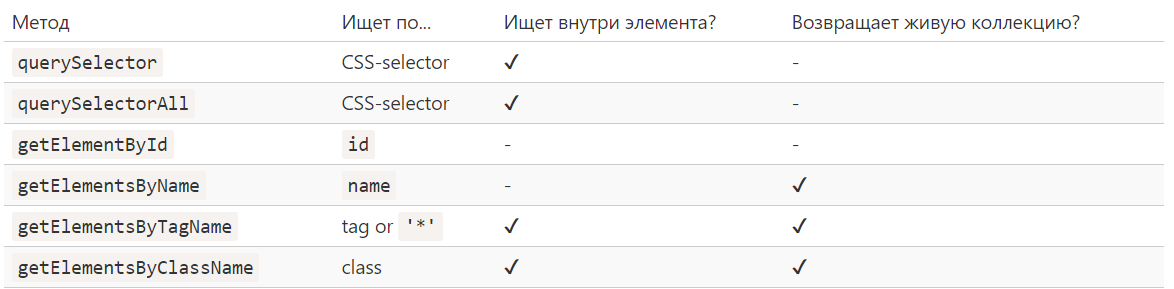
**.querySelector(css)** – работает как .querySelectorAll, но искать и возвращать будет не все элементы, а только первый

elem.matches(css) – проверяет, удовлетворяет ли elem css и возвращает true или false. Этот метод удобен, когда мы перебираем элементы и пытаемся выбрать те, которые нас интересуют



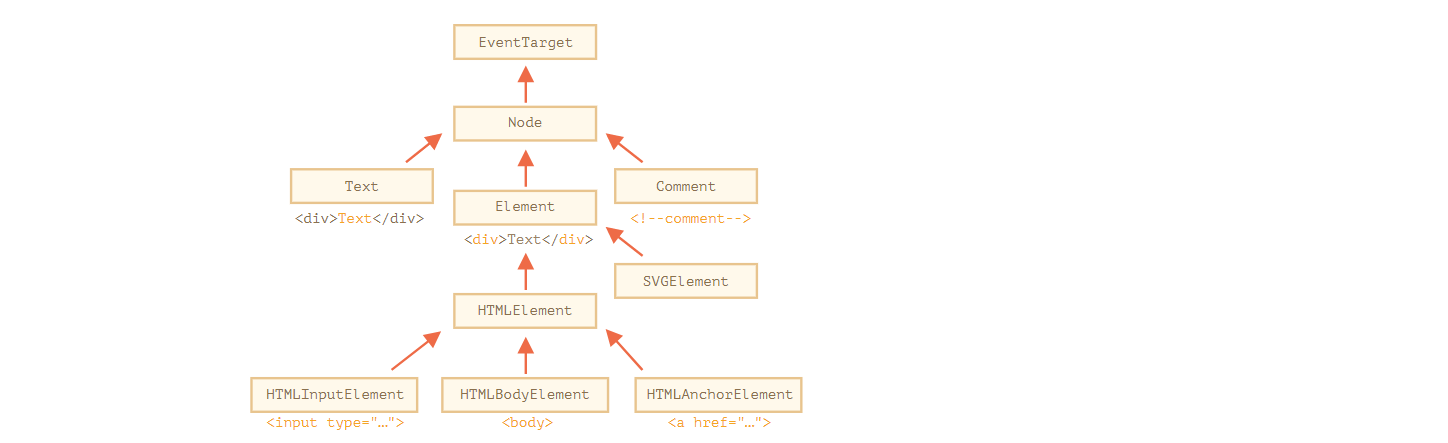
Нельзя менять значение напрямую, так как это коллекция

Работает

Итого

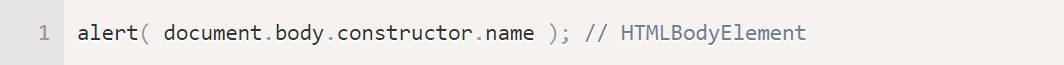
**Свойства узлов: тип, тег и содержимое  
Классы DOM-узлов**У разных DOM-узлов могут быть разные свойства, например у ссылки есть href, у поля ввода есть placeholder, а у кнопки есть type

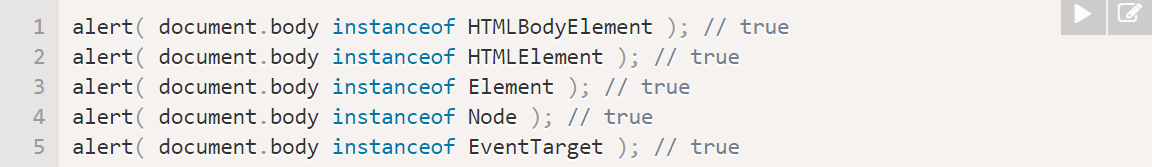
Несмотря на это, у них также есть общие методы и свойства, ведь все классы DOM-узлов образуют единую иерархию

Каждый DOM-узел принадлежит соответствующему встроенному классу  
Корнем иерархии является EventTarget, от него наследует Node и остальные DOM узлы

**Разберем эти классы подробнее:**  
**EventTarget** – это корневой класс. Объекты этого класса никогда не создаются. Он нужен для того, чтобы все элементы поддерживали события  
**Node** – класс, который служит основой для DOM-узлов. Объекты этого класса никогда не создаются. Он обеспечивает базовую функциональность parentNode, nextSibling, childNotes. Существует 3 класса, которые могут от него наследоваться: Text, Element, Comment  
**Element** – базовый класс для DOM-элементов. Он обеспечивает навигацию на уровне элементов: nextElementSibling, children, методы поиска: getElementsByTagName, querySelector. Браузер поддерживает не только HTML, но также и XML, SVG. Данный класс служит основой для 3 других классов: SVGElement, XMLElement, HTMLElement  
**HTMLElement** – базовый класс для остальных HTML-элементов. Он создает элементы типа HTMLInputElement (для поля ввода), HTMLAnchorElement (для ссылки) и тд

Таким образом, html-элемент (пр. input) проходит несколько наследований:  
Object -> EventTarget -> Node -> Element -> HTMLElement -> HTMLInputElement

Для того, чтобы узнать имя класса DOM-узла, обратимся к свойству constructor.name  


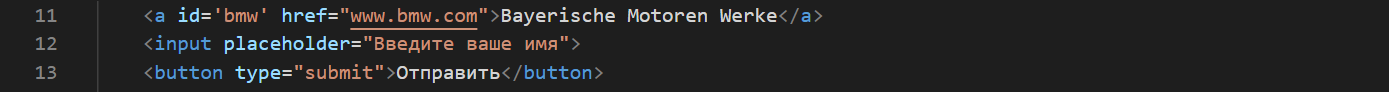
Проверить наследование можно также при помощи instanceOf  


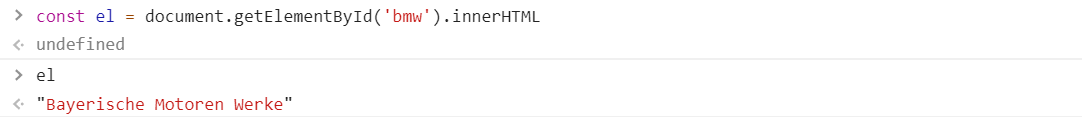
console.log – выводит элемент в виде DOM-дерева  
console.dir – выводит элемент в виде DOM-объекта

**Свойство nodeType**elem.nodeType == 1 – для узлов-элементов  
elem.nodeType == 2 – для узлов-атрибутов  
elem.nodeType == 3 – для текстовых узлов  
elem.nodeType == 8 – для узлов-комментариев  
elem.nodeType == 9 – для объектов документа  
elem.nodeType == 1 – для типа-документа

**Теги nodeName и tagName**Получив DOM-узел, можно узнать имя его тега из свойств nodeName и tagName  


Какая разница между tagName и nodeName?  
- свойство tagName есть только у элементов Element  
- свойство nodeName есть у любых узлов Node

**innerHTML**Рассмотрим код ниже

Выведем с помощью innerHTML содержимое id=«bmw» 

Таким образом innerHTML выводит HTML содержимое, которое находится по адресу

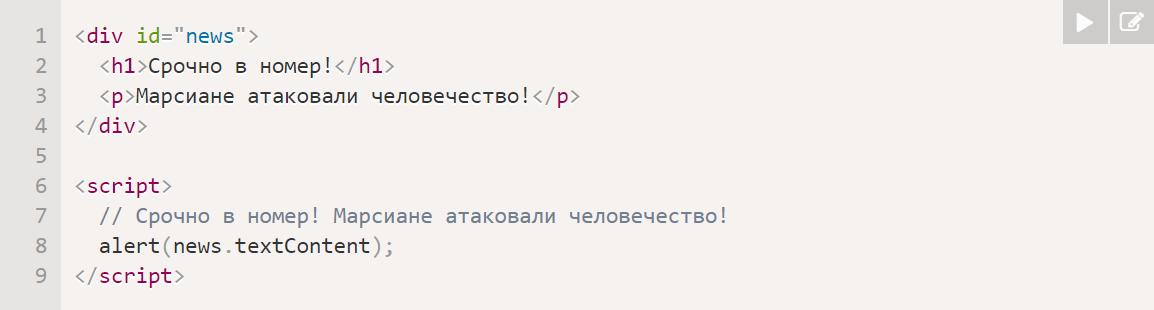
Однако мы можем перезаписать значение по этому адресу, при этом можем добавить даже теги  


Получаем  


**outerHTML**Данный метод в отличие от innerHTML вызывает не содержимое внутри тега, а все, что касается самого тега и внутри него

Таким образом данный код выведет  


Вот такой результат в консоль  


**textContent –** данный метод возвращает строку текстовых узлов

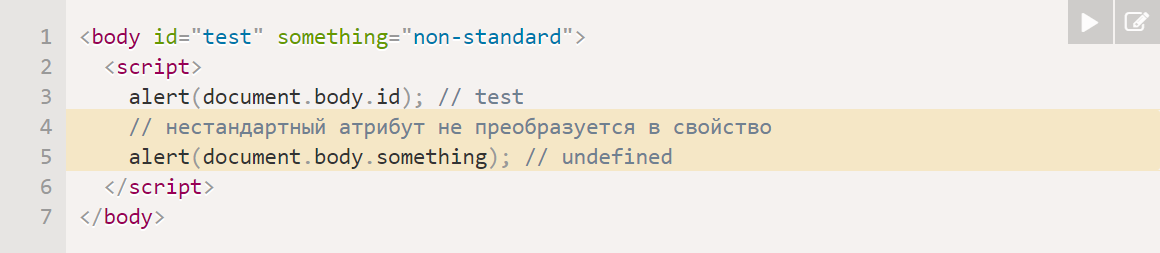
# **06.08.2020 5 часов Отчет: продолжаю изучать Документ**

**Атрибуты и свойства**Когда браузер загружает страницу, он «читает» html и генерирует из него DOM-объекты

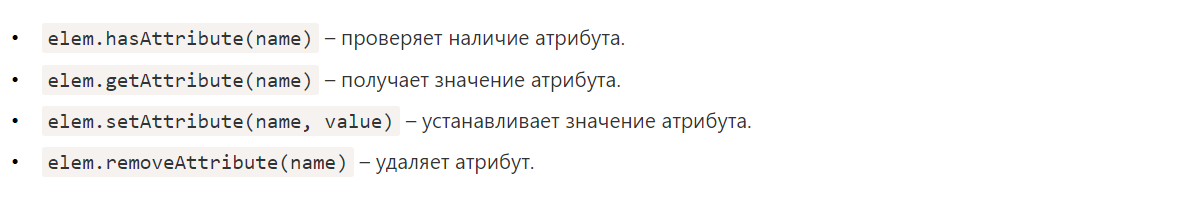
**DOM-свойства**DOM-узлы – обычные объекты, которые можно изменять

Таким образом, DOM-свойства и методы ведут себя также, как и обычные JS объекты:  
- им можно присваивать любые значения  
- они регистрозависимы (нужно писать elem.nodeType, а не elem.NodeTYpe)

**HTML-атрибуты**Когда браузер парсит html, чтобы создать DOM-объекты для тегов, он распознает стандартные атрибуты и создает DOM-свойства для них

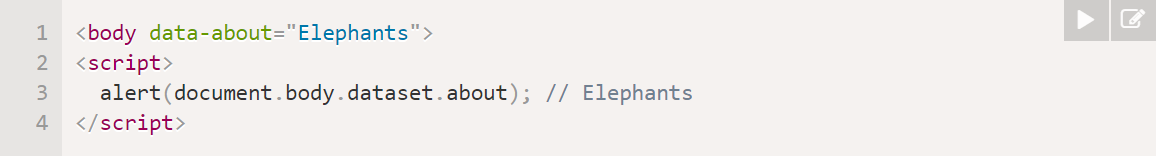
Таким образом, если у html тега есть id и нестандартный атрибут, то этот атрибут будет undefined  


Нестандартный атрибут – атрибут, которого не существует у тега

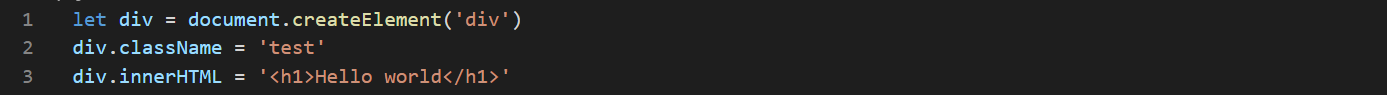
Для работы с атрибутами существуют методы  


У html атрибутов есть свои особенности  
- их имена регистронезависимы (т.е ID = id)  
- их значения всегда являются строками

В некоторых случаях атрибут может быть полезнее класса, так как атрибутом удобнее управлять  
Но с пользовательскими атрибутами могут возникнуть проблемы:  
иногда нам нужно использовать нестандартный атрибут, поэтому для того, чтобы не было неожиданных эффектов рекомендуется использовать атрибут data-

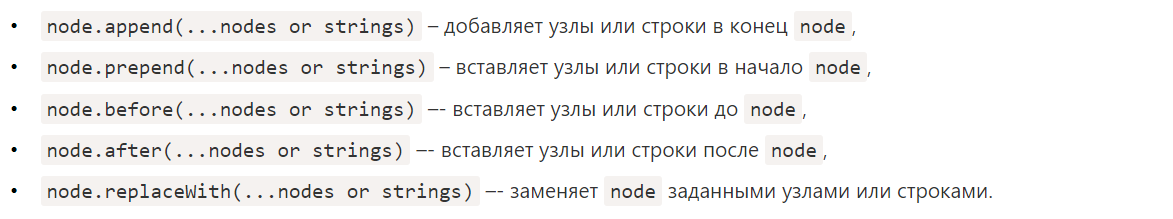
Все атрибуты, начинающиеся с префикса data-, зарезервированы для использования программистами. Они доступны в свойстве dataset

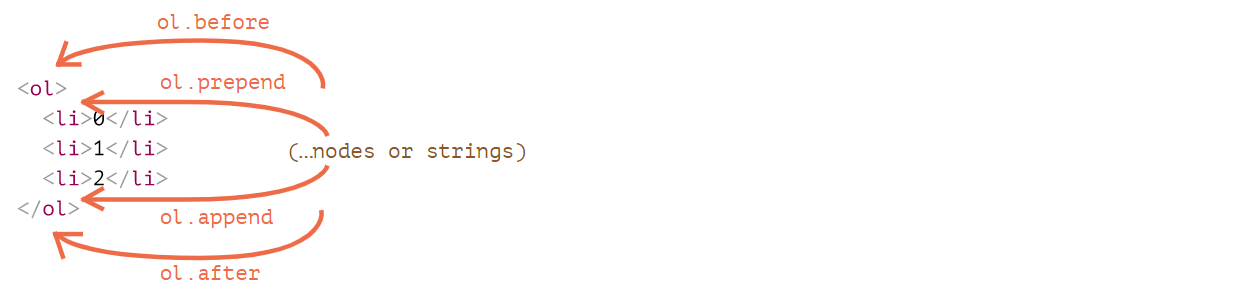
Атрибуты, которые состоят из нескольких слов, например data-show-info, становятся в JS в формат camel case: showInfo

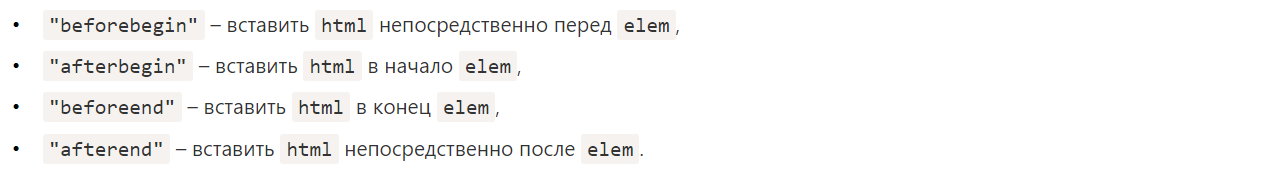
**Изменение документа**Создадим элемент div с классом text и значением h1 Hello world с помощью JS

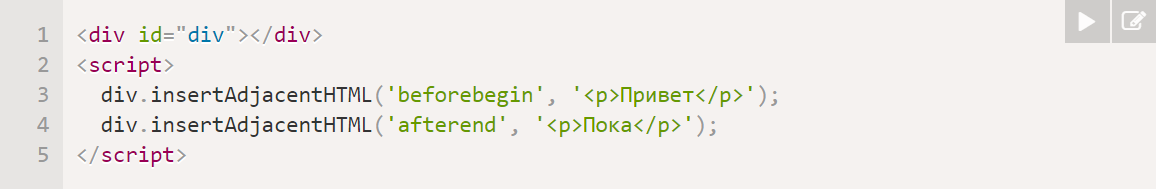
Элемент создан, однако он не отображается на странице, потому что его нужно вставить

Теперь работает

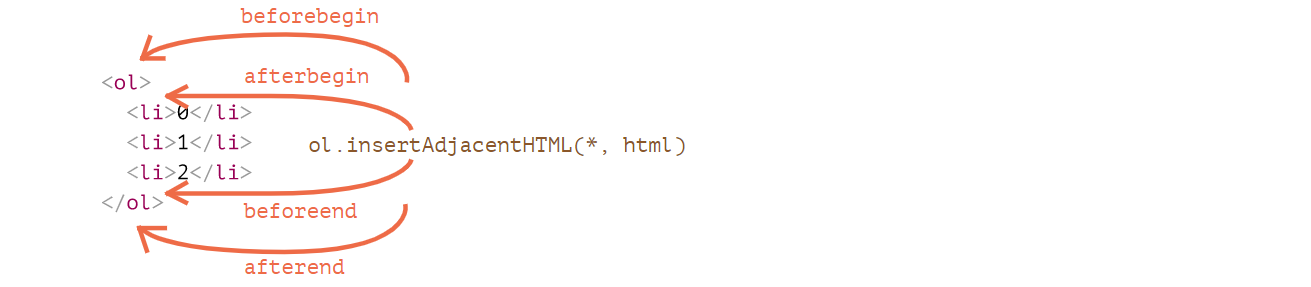
Методы для вставки элементов



**insertAdjacentHTML/Text/Element**Данный метод имеет синтаксис elem.insertAdjacentHTML(where, html)  
Where может принимать следующие значения  


Код ниже

Приводит к

Варианты вставки

На практике данный метод используется чаще, чем prepend и append. Большое преимущество данного метода заключается в том, что данный метод может вставлять html, причем код можно написать одной строчкой

Если нам нужно переместить элемент с одного места на другое, то нет необходимости его удалять  
Все методы вставки автоматически удаляют узлы со старых мест

**Клонирование узлов cloneNode**cloneNode(true) – метод, который создает «глубокий» клон элемента – со всеми атрибутами и дочерними элементами. При вызове cloneNode(false) клон будет без дочерних элементов

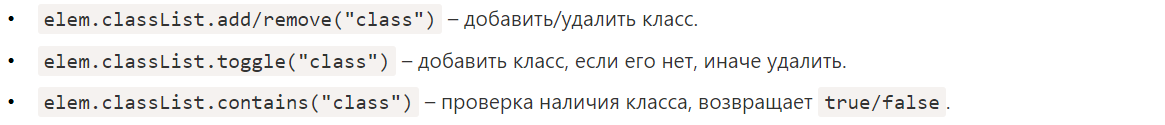
С помощью cloneNode можно изменить содержимое какого то блока:  
1) Клонировать этот блок  
2) Изменить его содержимое

**Устаревшие методы**

**12.08.2020  
2 часа Redux + 2 часа JavaScript  
Отчет: закончил изучать Essintials Part 3, закончил изучать Document JavaScript**

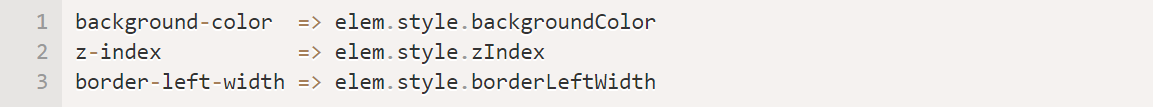
**JavaScript Document**  
**Styles and classes**Изменение класса – один из наиболее часто используемых действий в скриптах. Когда-то давно в JavaScript существовало ограничение: слово «class» было зарезервировано и оно не могло быть свойством объекта. Сейчас это ограничение отсутствует, но раньше писали elem.className

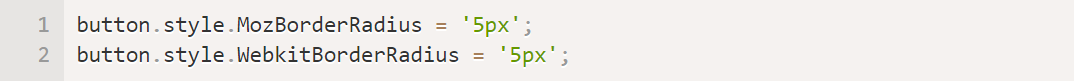
Сейчас elem.className заменяет всю строку с классами (иногда нам это нужно)  
Но часто мы хотим добавить или удалить один класс и используем elem.classList

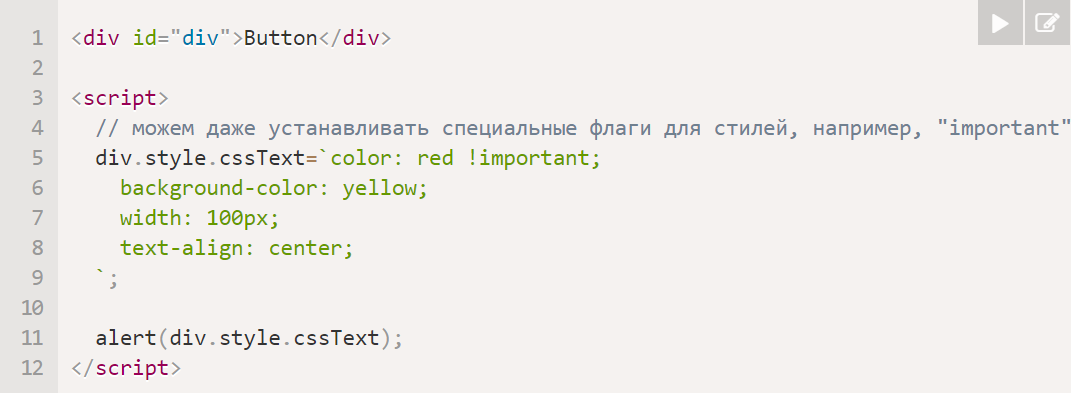
elem.classList – специальный объект с методами для добавления или удаления ОДНОГО класса  


classList является перебираемым, можно перечислить все классы при помощи for…of:  


**Element style**Свойство elem.style – объект, который соответствует тому, что написано в атрибуте «style»  
elem.style.width = «100px» работает также как наличие в style width: 100px

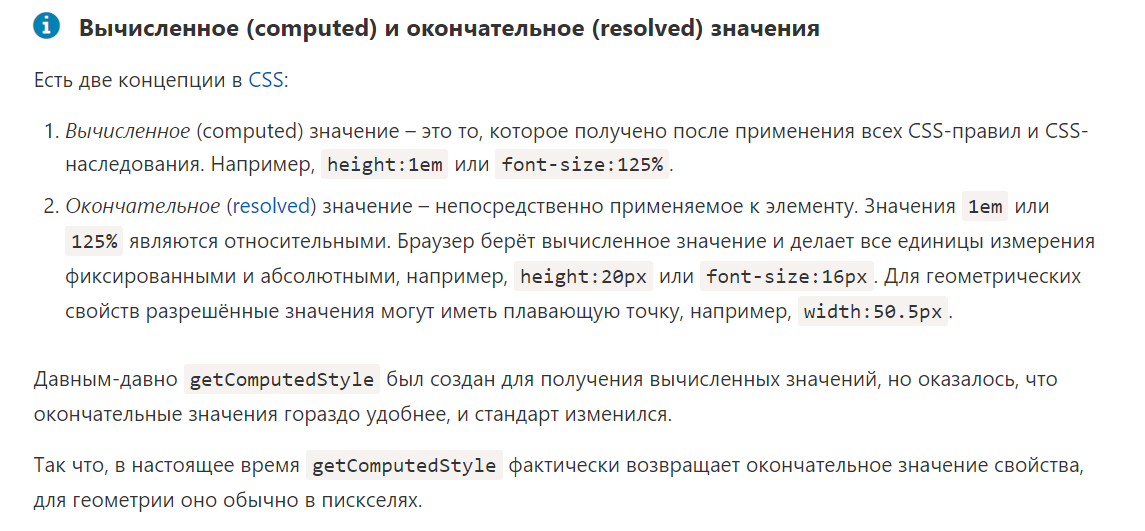
Для свойства из нескольких-слов стили приобретают camelCase:  


Стили с браузерным префиксом преобразуются таким же образом, только дефис = Большая буква

Для задания нескольких стилей в одной строке используется свойство style.cssText: 

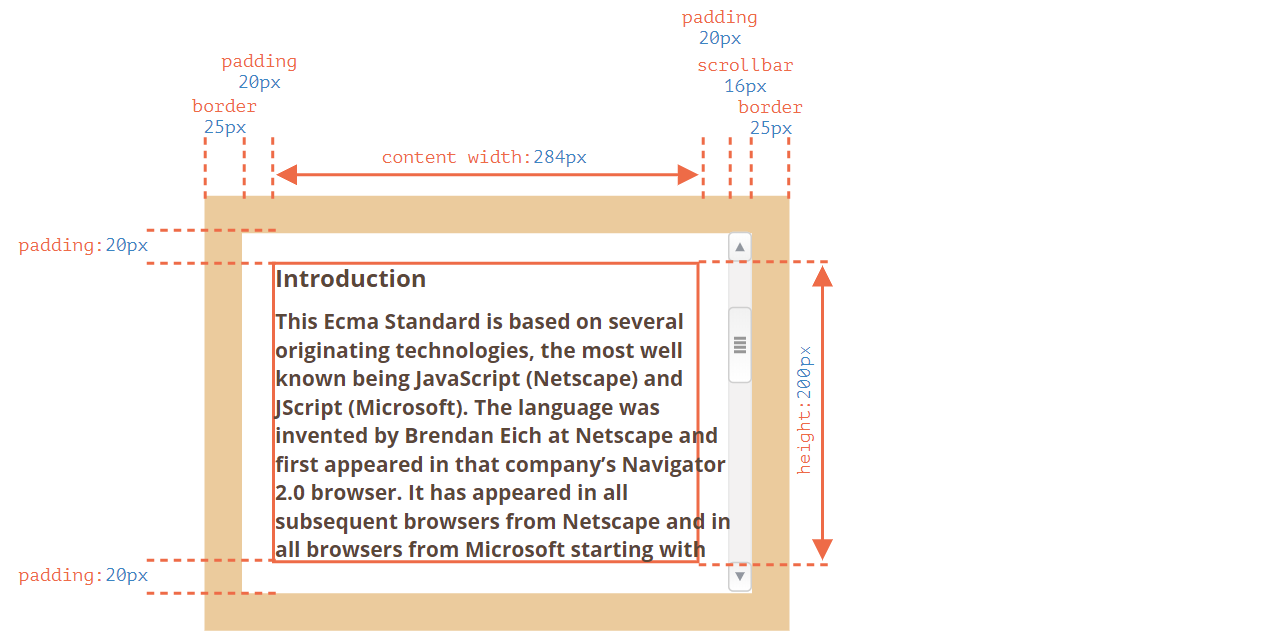
**Чтение стилей**Для того, чтобы прочитаь стили существует метод getComputedStyle 

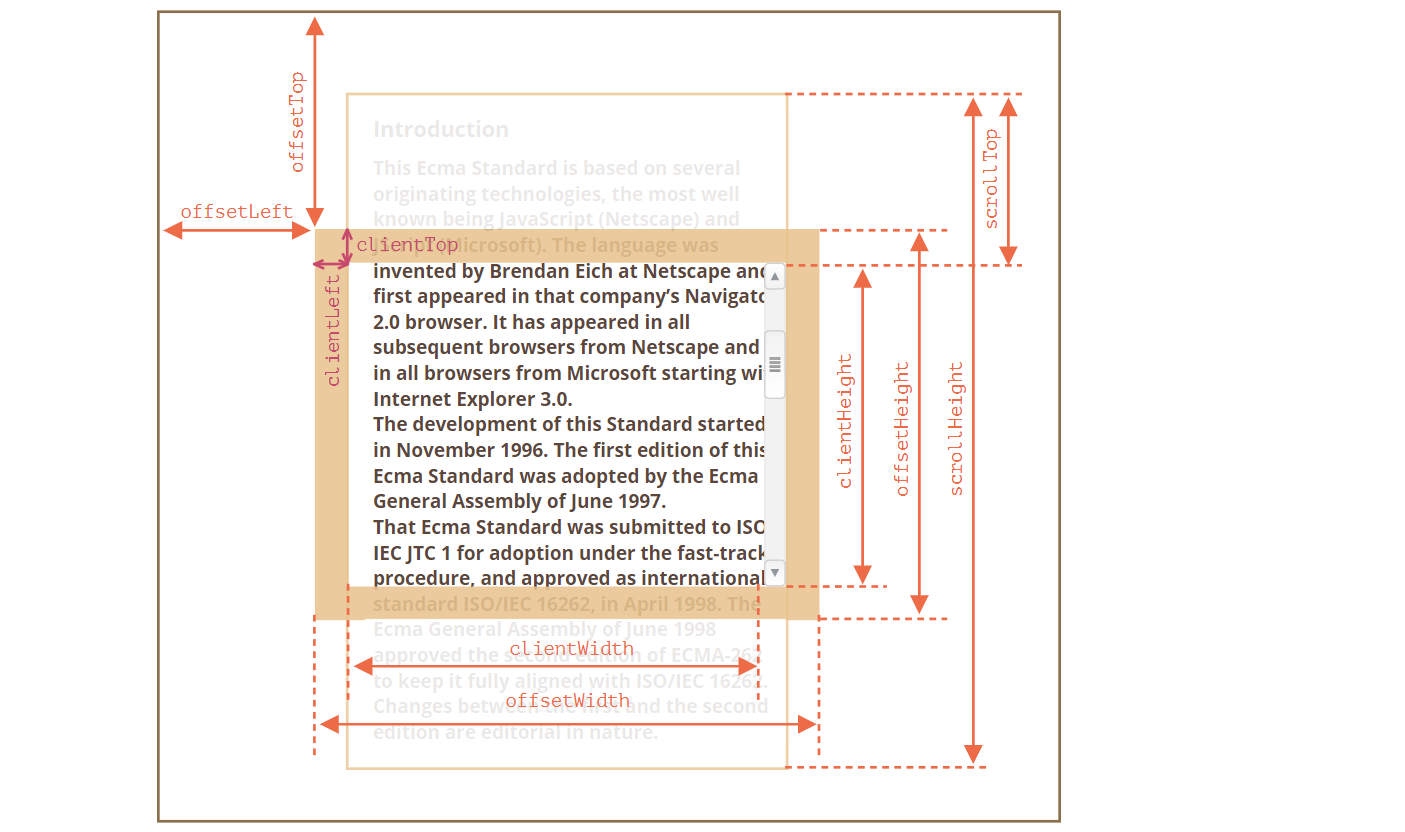
Element – элемент, значение для которого нужно получить  
Pseudo – указывается, если нужен стиль псевдоэлемента, например ::before. Пустая строка или отсутствие аргумента означают сам элемент

Результат вызова – объект со стилями, похожий на elem.style, но с учетом всех CSS-классов  

**Размеры и прокрутка элементов**Рассмотрим простой пример: 

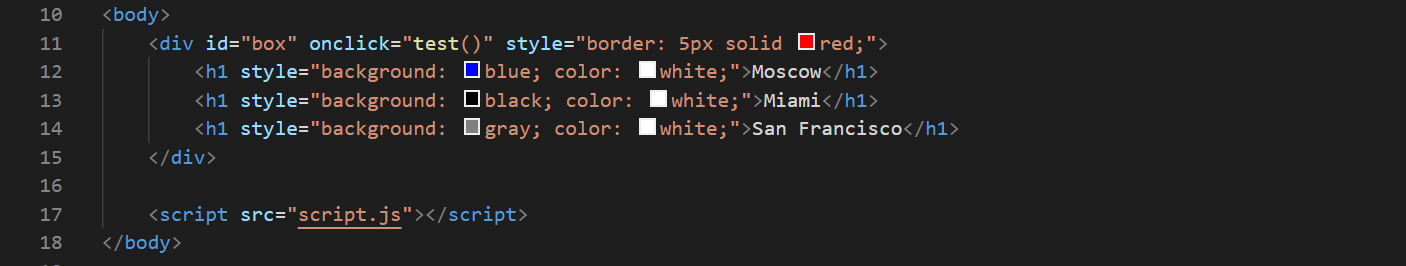
У элемента есть рамка (border), внутренний отступ (padding) и прокрутка, но нет внешних отступов (margin), потому что они не являются частью элемента. Для них нет особых JS-свойств

Результат выглядит так: 

А вот и общая картина с геометрическими свойствами:   
Значениями свойств являются числа, подразумевается, что они в пикселях

**13.08.2020  
5 часов  
Отчет: начал изучать введение в событие**

**Введение в события**Событие – это сигнал от браузера, что что-то произошло  
Событию можно назначить обработчик, то есть функцию, которая сработает, как только событие произошло

Первый способ задать событие (в HTML):

Второй способ (в JS):  

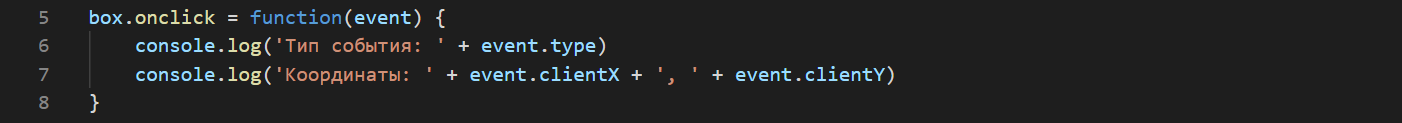

Третий способ: 

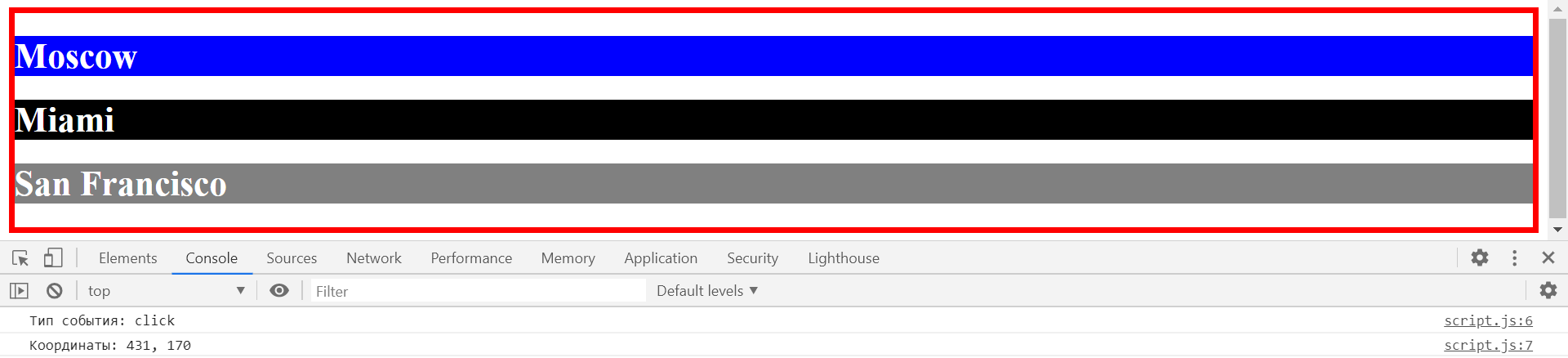
У первых двух способов есть существенный недостаток: нельзя повесить несколько обработчиков на одно событие. А в третьем способе мы можем делать это сколько угодно

Обработчики некоторых событий можно назначить только через addEventListener, например DOMContentLoaded, которое срабатывает, когда завершена загрузка и построение DOM

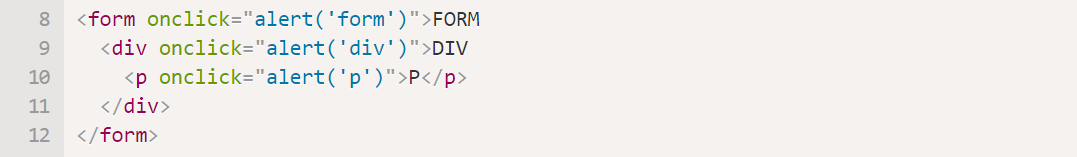
**Объект события**  
Чтобы обработать событие, могут понадобиться детали произошедшего: не просто клик, или нажатие клавиши, а координаты указателя мыши, какая клавиша нажата и тд

Когда происходит событие, браузер создает объект события, записывая в него детали и передает его в качестве аргумента функции-обработчику





**Всплытие и погружение**  
Вспылтие работает очень просто:  
Когда на элементе происходит событие, обработчики сначала срабатывают на нем, потом на его родители, затем выше и так далее, вверх по цепочке предков

Например, в коде ниже 

При нажатии на p произойдет следующий алгоритм:  
1) сработает событие на p  
2) сработает событие на div  
3) сработает событие на форму

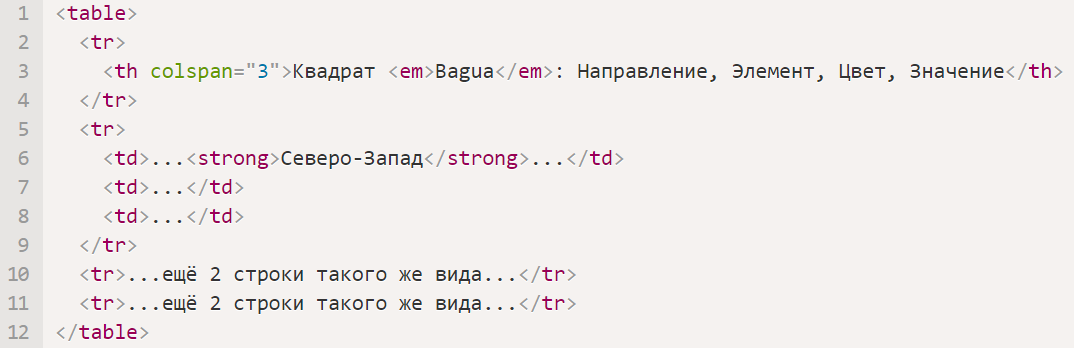
При нажатии на div:  
1) сработает событие на div  
2) сработает событие на форму

Это и есть процесс всплытия. Почти все события всплывают (кроме focus)

**Делегирование событий**  
Всплытие и перехват позволяют организовать один из самых важных приемов разработки – делегирование

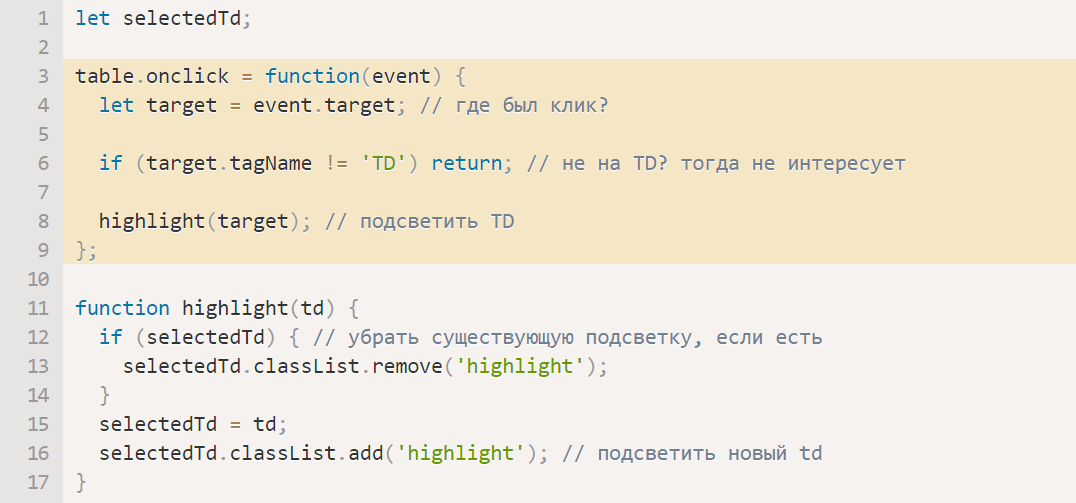
Если у нас есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы назначать обработчик каждому, мы ставим один обработчик на их общего предка. Из него можно получить event.target, понять на каком именно потоке произошло событие и обработать его

Рассмотрим следующую таблицу: 

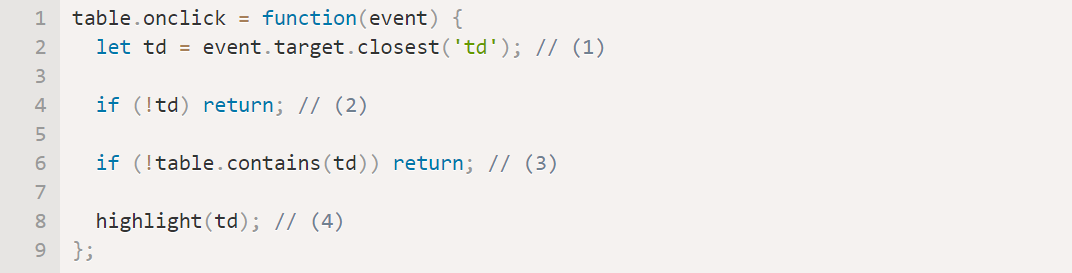
Ее реализация в HTML: 

На данный момент таблица имеет 9 ячеек. Это не так много. Мы можем привязать обработчик событий к каждой ячейке. Но, что делать если ячеек намного больше, например 500

Вместо того, чтобы назначить обработчик click для каждой ячейки, мы повесим единый обработчик на всю таблицу



У текущей версии кода есть недостаток:  
клик может быть совершен не на теге td а внутри него

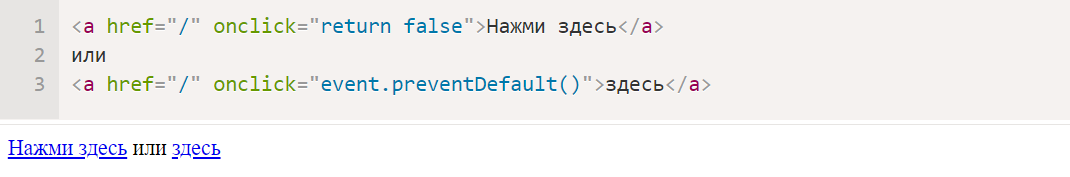
Внутри обработчика table.onclick мы должны разобраться по event.target был клик внутри td или нет. Вот улучшенный код: 

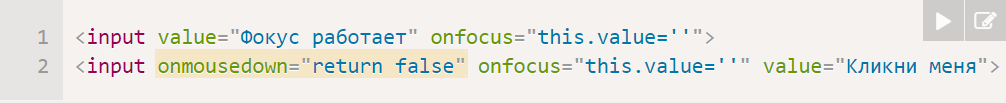
Преимущества делегирования:  
- упрощает процесс инициализации и экономит память: не нужно вешать много обработчиков  
- меньше кода: при добавлении и удалении элементов не нужно менять обработчики  
- удобство изменений DOM: можно массово добавлять и удалять элементы

Ограничения делегирования:  
- событие должно всплывать. Некоторые события этого не делают  
- создает нагрузку на браузер, т.к обработчик запускается, когда событие происходит в любом месте контейнера, не обязательно на элементах, которые нам интересны

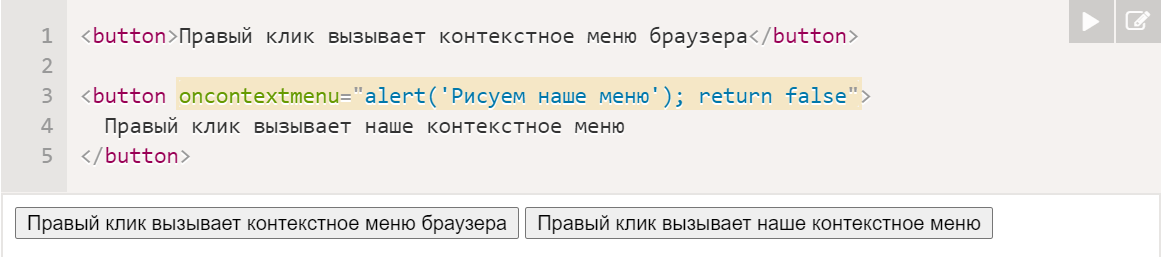
**Действия браузера по умолчанию**  
Многие события автоматически влекут за собой действие браузера, например:  
- клик по ссылке инициализирует переход на новый URL  
- нажатие на кнопку отправить в форме – отсылку на ее сервер  
- зажатие кнопки мыши над текстом и ее движение – инициализирует выделение

Есть два способа отменить действие браузера:  
- основной способ – воспользоваться объектом event. Для отмены действия браузера существует стандартный метод event.preventDefault()  
- если же обработчик назначен через onsomething (не через addEventListener) то также можно вернуть false из обработчика

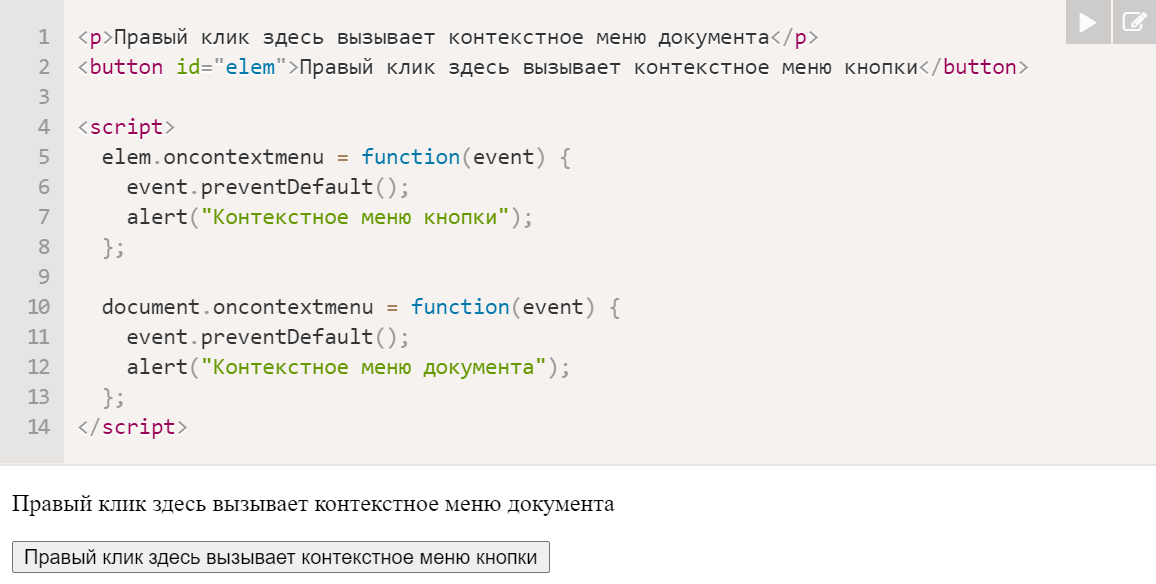
В примере ниже переход по ссылке не произойдет: 

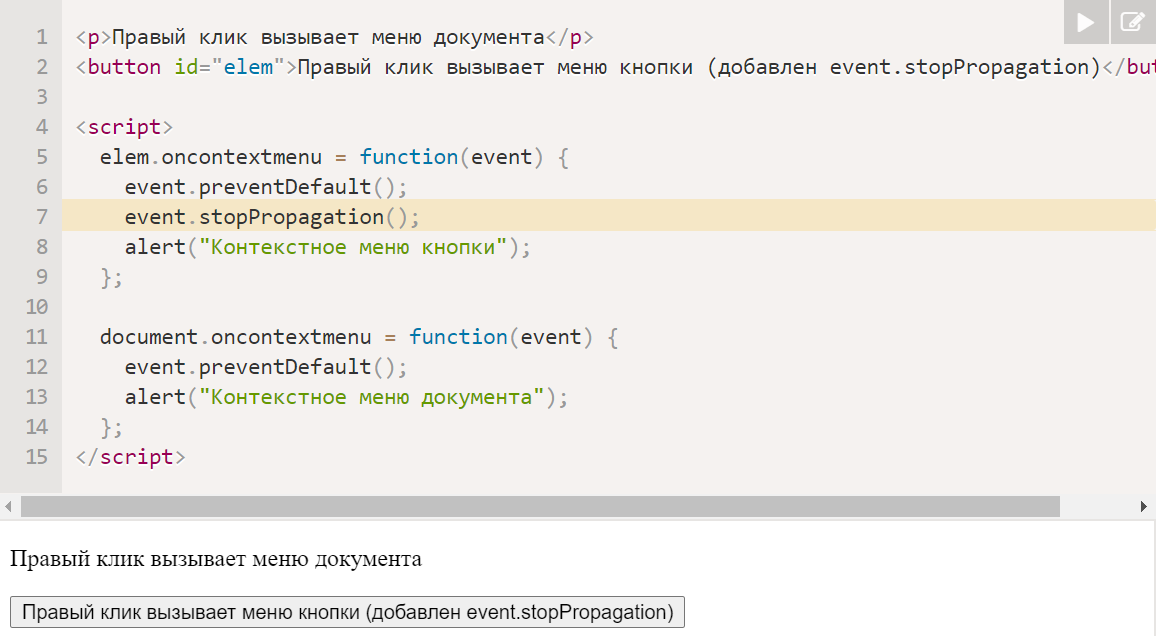
Некоторые события естественным образом вытекают друг из друга. Если мы отменим первое событие, то последующие не возникнут. Например, событие mousedown для поля input приводит к фокусировке на нем и запускает событие focus. Если мы отменим событие mousedown для input, то фокусировки происходить не будет 

В примере выше фокус на первом инпуте работает, а на втором нет. Однако если дотянуться до второго инпута Tabом, то тогда фокус будет работать тоже

**event.PreventDefault**  
Рассмотрим пример с нажатием правой кнопки мыши на элемент (oncontextmenu). По умолчанию браузер при событии contextmenu показывает контекстное меню, однако мы может предотвратить это действие, и сделать что-то свое: 

Таким образом, кнопка слева покажет контекстное меню  
А кнопка справа выведет alert

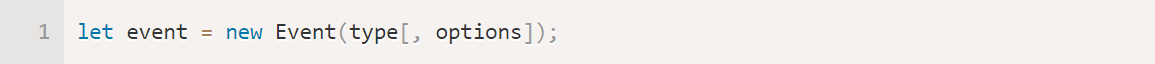
Теперь сделаем так, чтобы при нажатии на правую кнопку мыши нигде не выходило контекстное меню 

При клике правой кнопкой мыши на кнопку мы получим два alert. Для того, чтобы исправить это нам нужно вызвать метод event.stopPropagation после event.PreventDefault на кнопке  


**Генерация пользовательских событий**  
Можно не только назначать обработчики, но и генерировать события из JS-кода

Пользовательские события могут быть использованы при создании графических компонент. Например, корневой элемент меню, реализованный с помощью JS, может генерировать события, которые относятся к этому меню: open, select и тд. А другой код может слушать эти события и узнавать, что происходит с меню

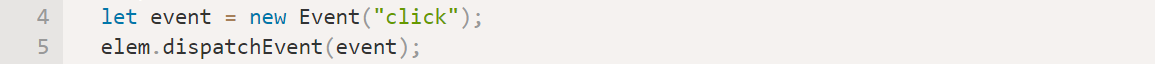
**Конструктор Event**  
Встроенные классы для событий формируют иерархию аналогично классам для DOM-элементов. Ее корнем является встроенный класс Event

Событие встроенного класса можно создать так:  


Type – тип события (click, mouseover, keyup, или придуманное code-end)  
options – объект с двумя необязательными свойствами  
 1) bubbles: true/false (если true, тогда событие всплывает)  
 2) cancelable: true/false (если true, тогда можно отменить действие по умолчанию)  
 По умолчанию эти свойства установлены в false

**Метод dispatchEvent**После того, как объект события создан, мы должны запустить его на элементе, вызвав метод elem.dispatchEvent(event)

Затем обработчики отреагируют на него, как будто это обычное браузерное событие. Если при создании указан флаг bubbles, то оно будет всплывать

В примере ниже событие click инициализируется JavaScript-кодом так, будто кликнули по кнопке:  
  


**event.isTrusted**  
С помощью данного свойства можно легко отличить встроенное событие от созданного. Если данное свойство принимает значение true - то оно встроенное, если false – созданное

**Пример всплытия**  
Мы можем создать свое собственное событие с именем hello и поймать его на document. Все, что нужно сделать, это установить флаг bubbles в true  


Правила при создании собственных событий:  
- нужно использовать addEventListener, т.к onevent существуют только для встроенных событий  
- обязательно нужно передать флаг bubbles: true, иначе событие всплывать не будет