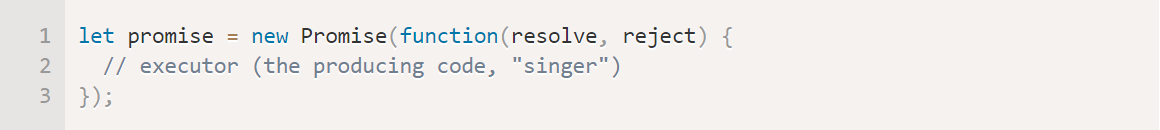
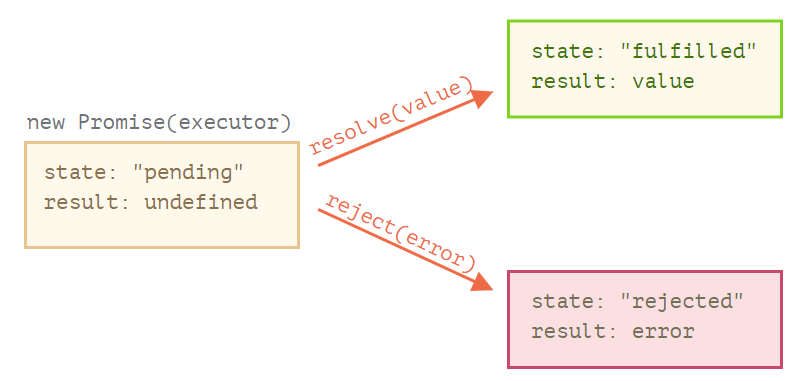
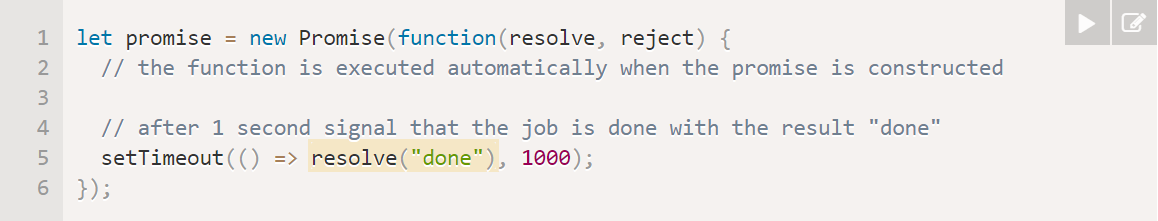
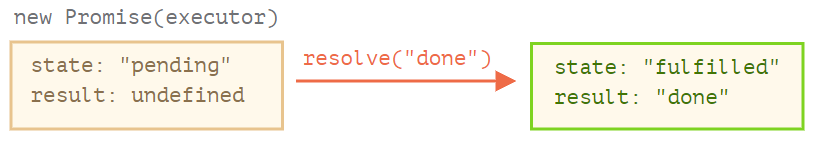
# **03.08.2020 5 часов Отчет: начал изучать promises 1-4**

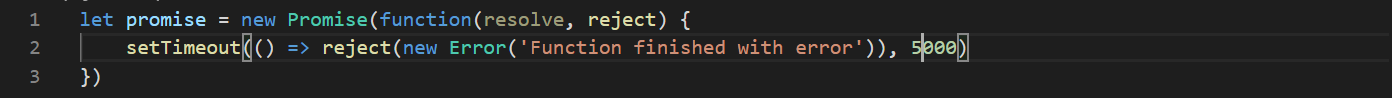
**Promises**Синтаксис создания:   
Функция, которая передается в promise, называется executor. Когда promisе создался, исполнитель запускается автоматически. Resolve и reject – колбэки, которые предоставляет promise

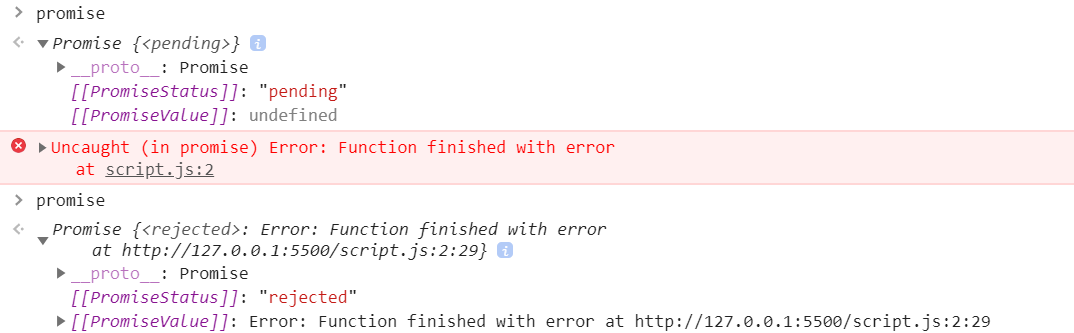
Resolve(value) вызывается, если работа колбэка завершилась успешно с результатом value  
Reject(error) вызывается, если есть ошибки, error – данные об ошибке

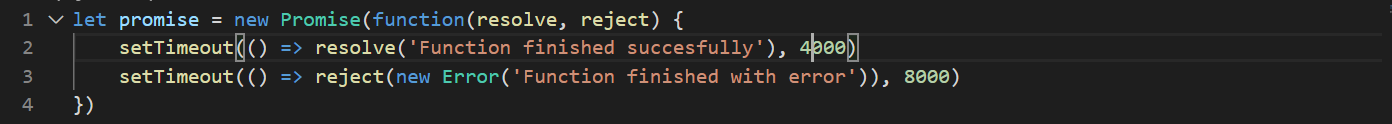
У объекта promise, возвращаемого конструктором new Promise, есть внтуренние свойства:  


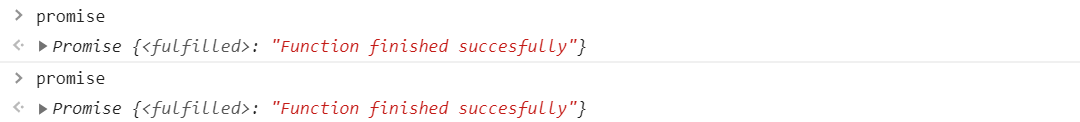
В примере ниже, мы создаем promise, причем, запустив код, можем наблюдать две вещи:  
1) Функция-исполнитель запускается сразу же при вызове new Promise  
2) Исполнитель получает две встроенные функции resolve и reject

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 

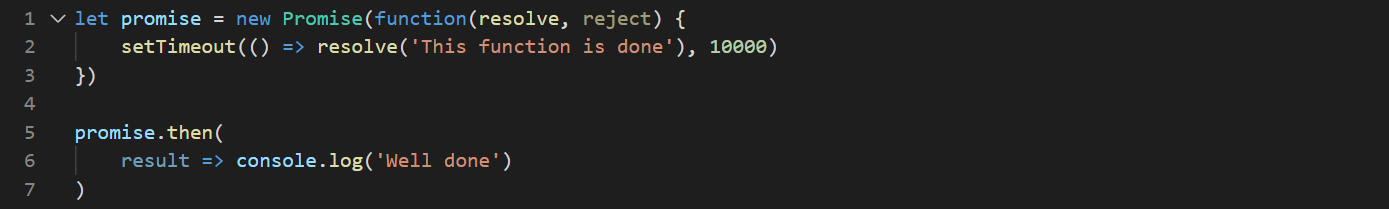
Также хотелось бы отметить, что executor может вернуть только resolve или только reject: 

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


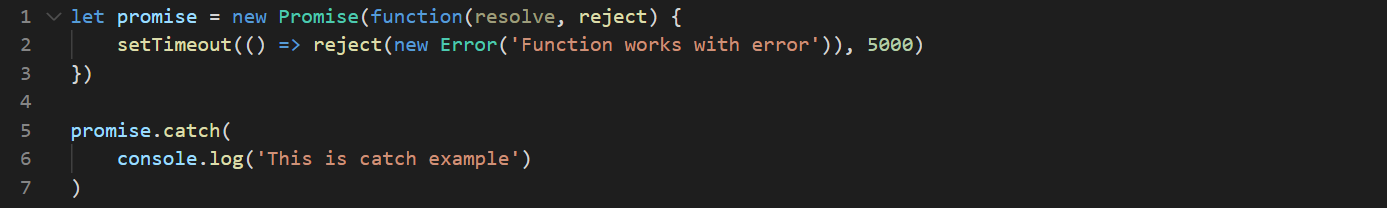
Свойства state и result – это внутренние свойства объекта promise, к нему мы не имеем прямого доступа. Для обработки результата следует использовать методы .then .catch .finaly

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

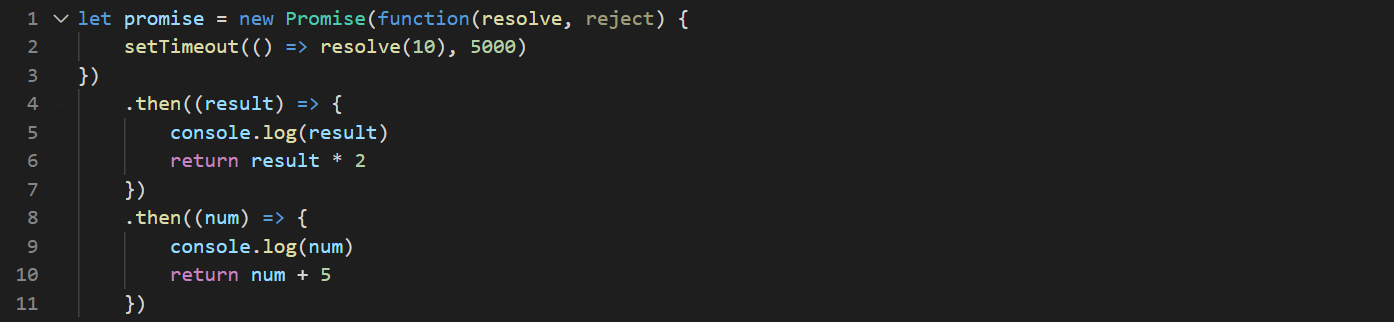
Рассмотрим первый случай, когда промис перешел в состояние «выполнен успешно»   
Что делает данный код?  
1) Изначально promise создается с помощью конструктора new Promise и имеет status: pending, value: undefined  
2) Через 10 секунд вызывается функция resolve, которая меняет свойства promise на status: 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

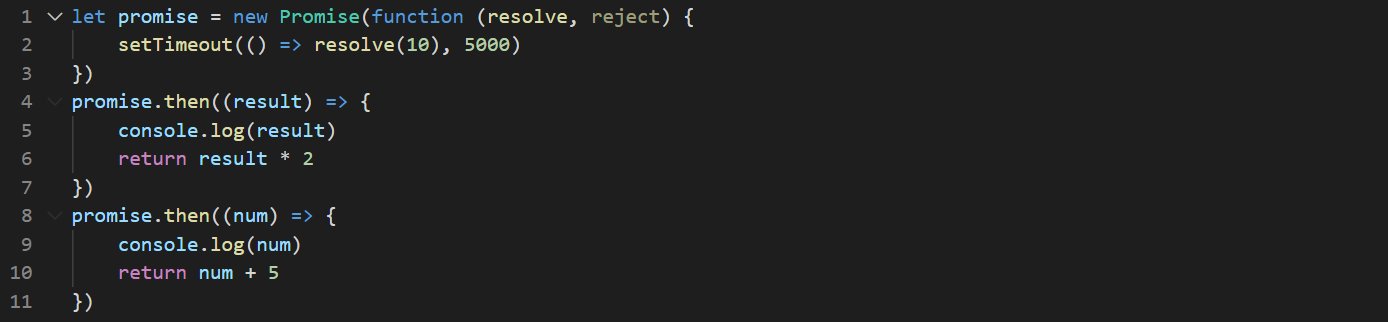
Если нам нужно, чтобы метод .then реагировал также на ошибку, то после 6ой строчки можно вставить следующий код: error => console.log(error)

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

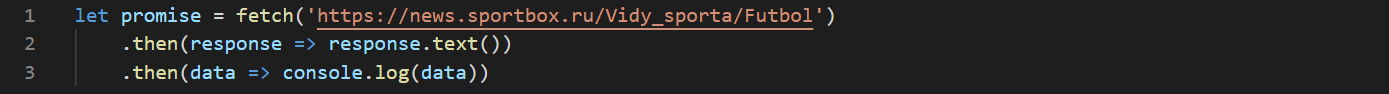
.finally – метод, который выполнится в любом случае, в не зависимости от того, как завершил работу promise

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

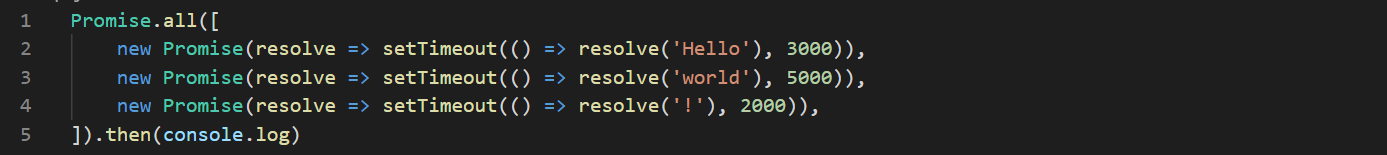
Что будет если ошибка не будет обработана? То есть если будет ошибка, но не будет указано метода .finally или .catch который обработал бы ошибку?  
В таких случаях движок JS отслеживает такие ситуации и генерирует глобальную ошибку

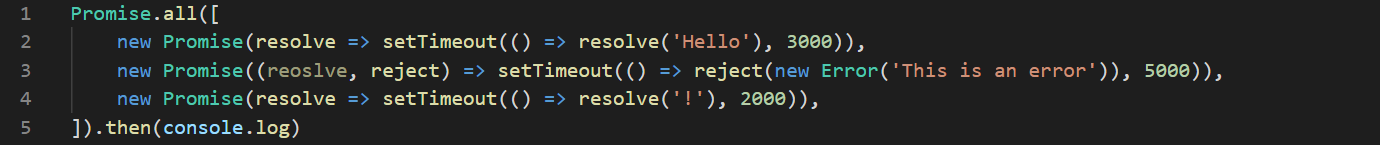
В браузере можно поймать ошибки, используя событие unhandlerejection  
Если происходит ошибка, и отсутствует ее обработчик, то генерируется событие unhadlerejection и соответствующий объект event, который содержит информацию об ошибке

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

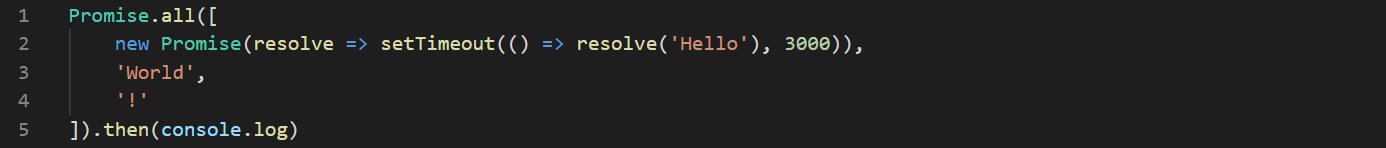
**Promise API**В классе Promise есть 5 статических методов

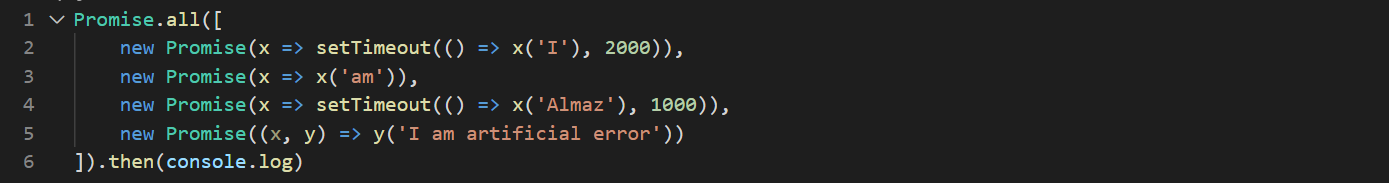
**Promise.all** – используется, когда нужно запустить множество promise параллельно и дождаться, пока они все выполнятся

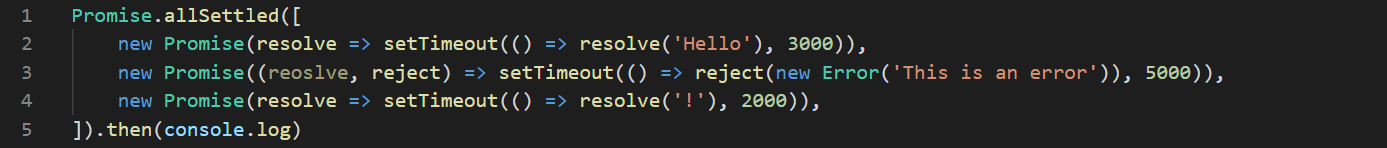
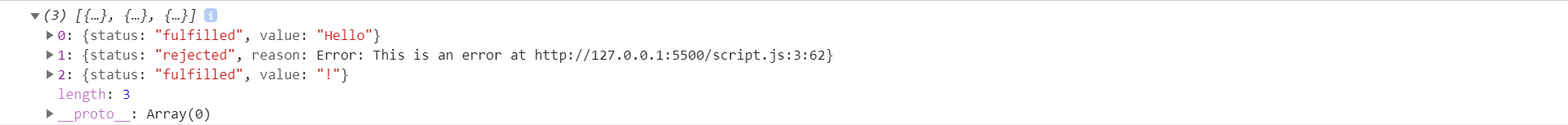
Метод Promise.all принимает массив промисов и возвращает новый массив  
Новый промис завершится, когда завершится весь переданный список промисов, и его результатом будет массив их результатов   

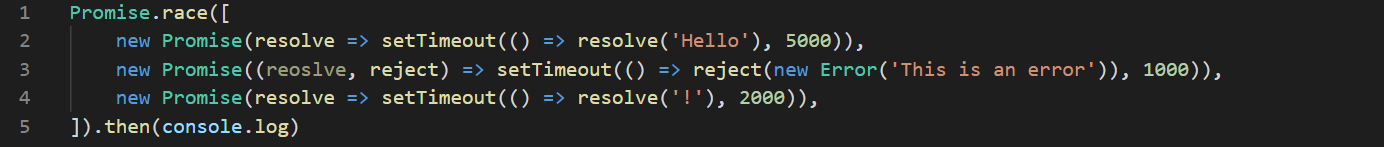

Если хотя бы один промис завершается с ошибкой, то Promise.all возвращает эту ошибку  

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

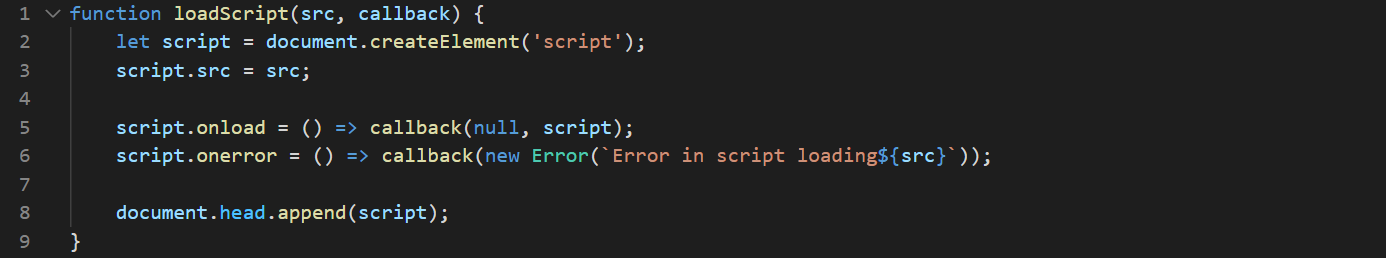
**Promise.allSettled** в отличие от Promise.all не будет прекращать свою работу в случае обнаружения ошибки в объекте Promise. Promise.allSettled полностью обработает все промисы, вернет массив объектов с ключами status, value  
 

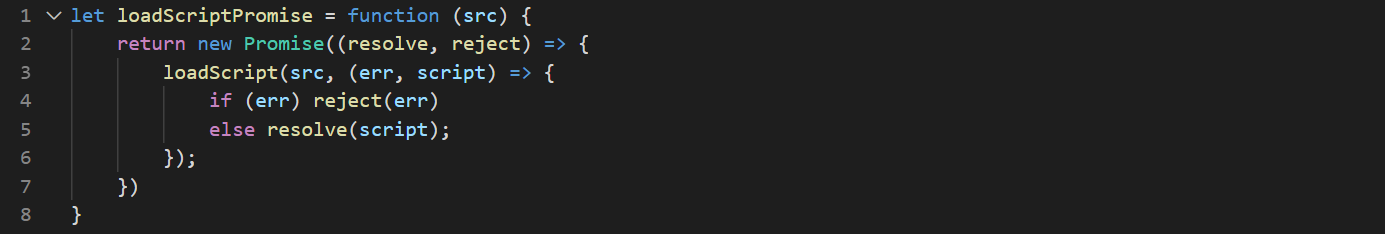
**Promise.race** – возвращает результат промиса, который завершил работу самый первый   
В данном примере Promise.race вернет ошибку, так как этот промис завершится быстрее всех  
 

**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

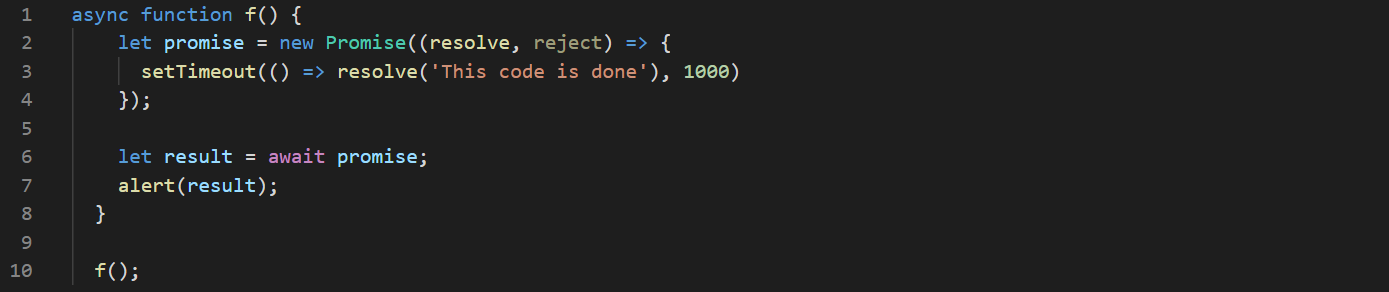
В спецификации сказано:  
- очередь выполняется по принципу «первый пришел – первый ушел». То есть задачи, попавшие первыми, выполняются тоже первыми  
- выполнение задачи происходит только в том случае, если больше ничего не запущено. Или, проще говоря, когда промис выполнен, его обработчики .then .catch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

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

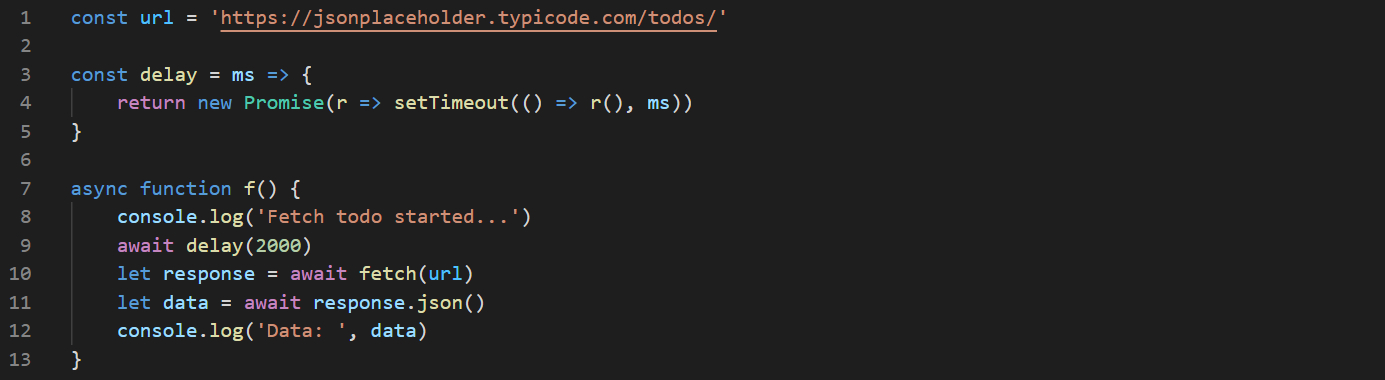

**Async/await  
Async –** ключевое слово, которое ставится перед функцией. У слова async один простой смысл – эта фукнция всегда возвращает промис.

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: 

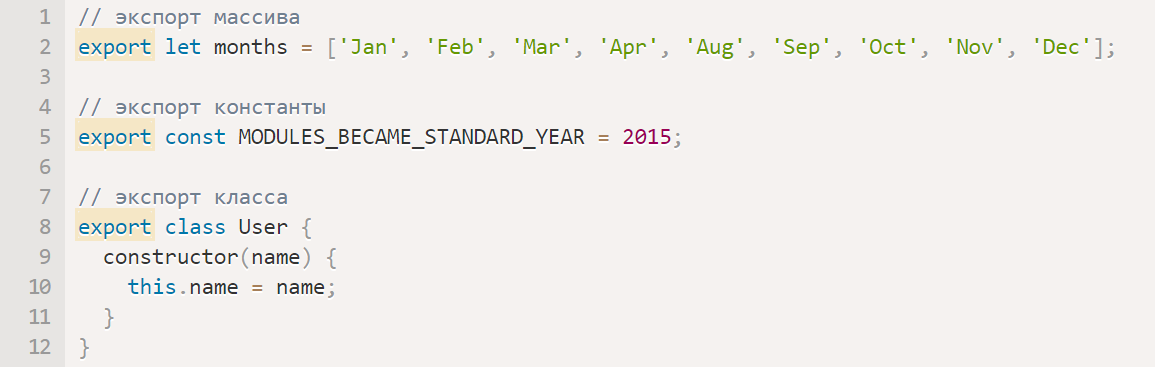
А здесь мы импортируем функцию sayHi с модуля sayHi.js

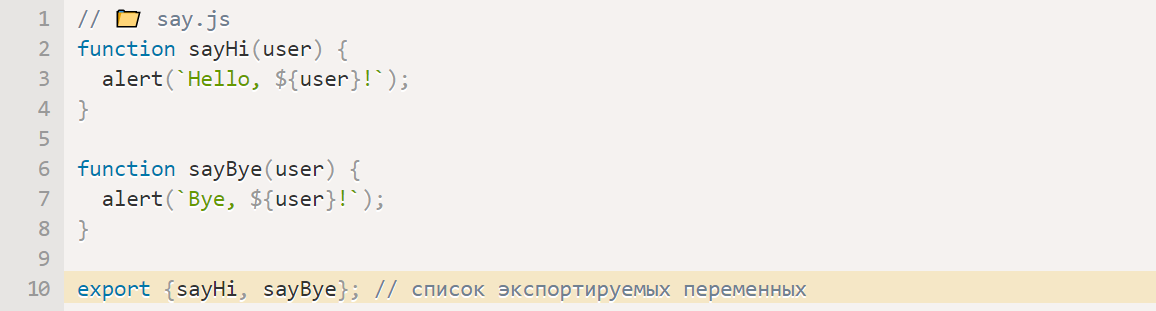
Чтобы пользоваться модулями, достаточно при объявлении скрипта указать type=«module»  

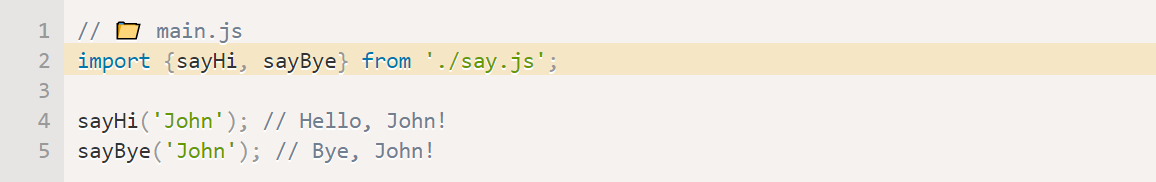

Отличия модулей от обычных скриптов:  
1) В модулях всегда используется режим use-strict  
2) Модули имеют свою область видимости. Т.е переменные и функции, объявленные внутри модулей не доступны для скриптов. Если нам нужно сделать глобальную переменную для всего документа, то мы ее можем объявить с 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

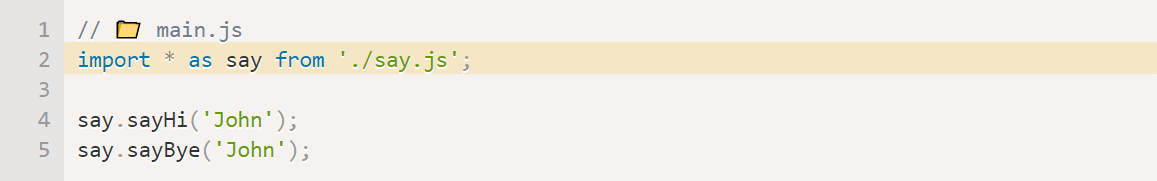
Особенности в браузерах:  
1) Модули всегда выполняются в режиме defer, т.е   
- загрузка внешних модулей не блокирует html  
- модули всегда ожидают полной загрузки html, только потом выполняются  
- сохраняется относительный порядок скриптов  
2) Атрибут async работает во встроенных скриптах, т.е для 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

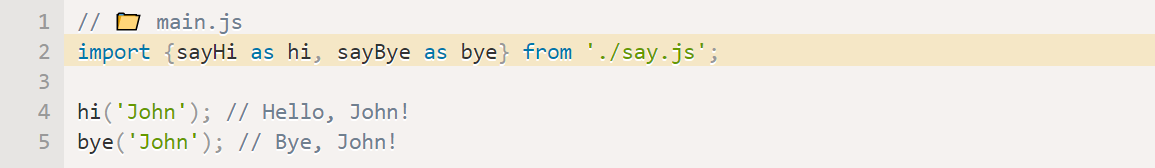
Экспортировать можно до объявления:  


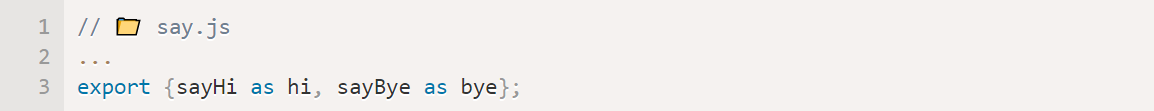
И отдельно от объявления: 

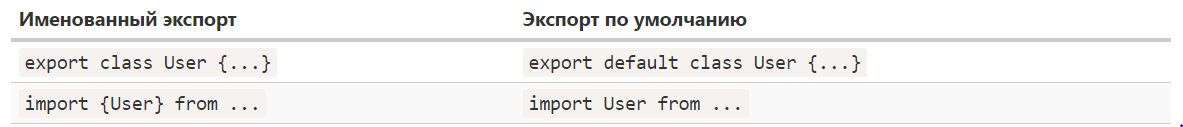
Для того, чтобы импортировать конкретные файлы, нам следует их указать при импорте:  


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  


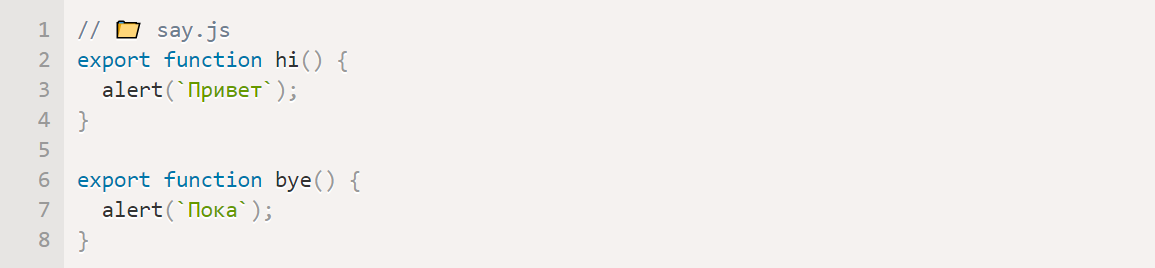
Таким же образом мы их можем и экспортировать  


**Export default**

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

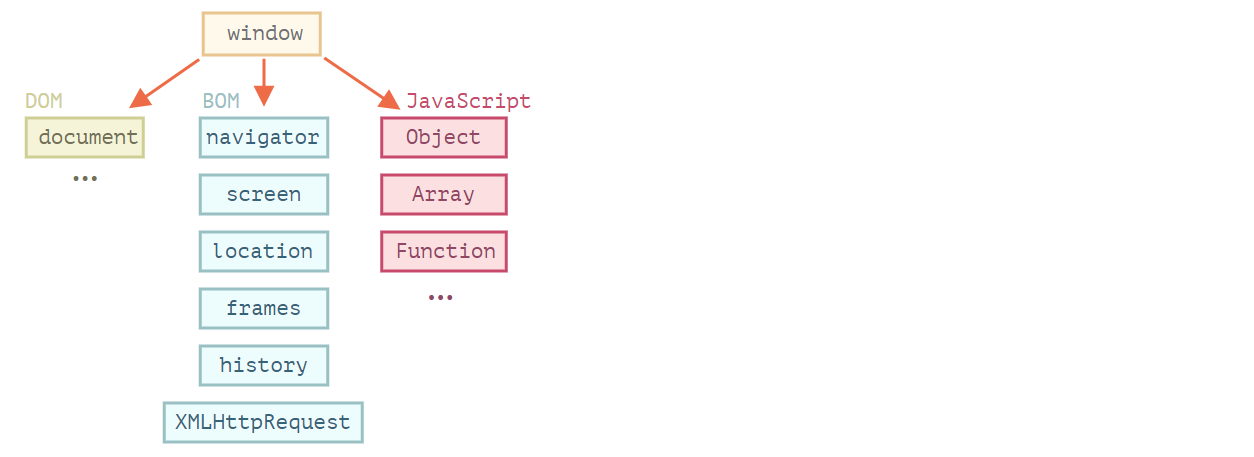
**Динамические импорты**Те инструкции, которые мы рассматривали в предыдущих главах, называются статическими. Они имеют достаточно простой и строгий синтаксис:  
1) Путь к импорту должен быть строковым примЛитивом и не может быть вызовом функции  
2) Нельзя делать импорт в зависимости от условий

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

Если у нас есть такой модуль say.js

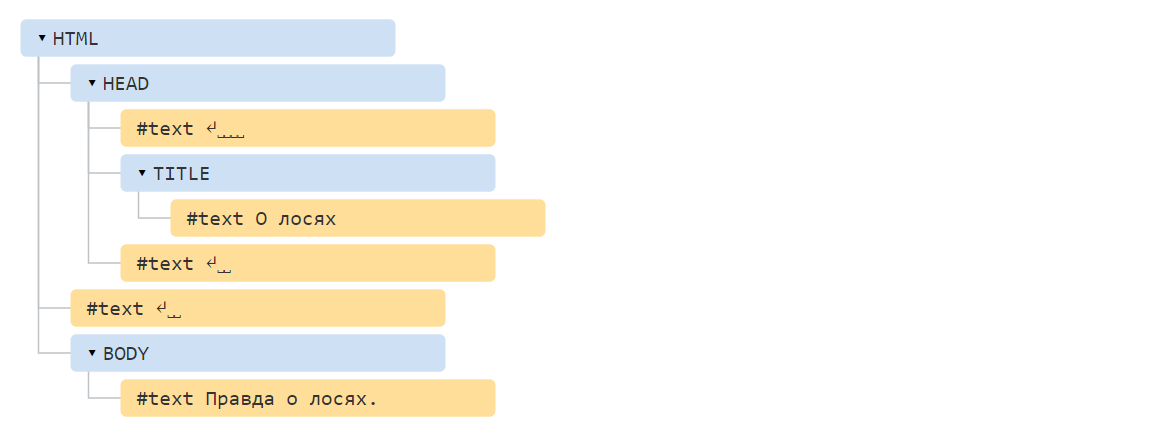
То динамический импорт может выглядеть вот так  

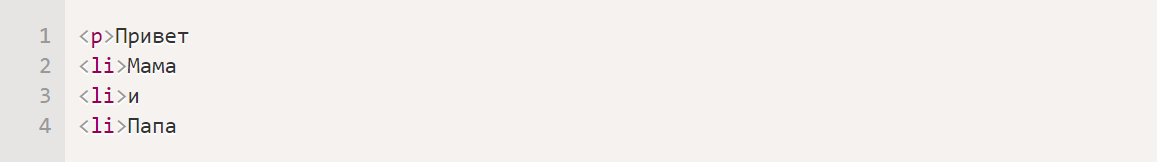

Динамический импорт работает в обычных скриптах и не требует указания type=module

**Браузерное окружение, спецификации**

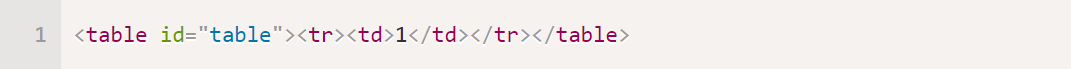
Стандарты:  
DOM – описывает структуру документа, манипуляции с контентом и события  
CSSOM – описывает файлы стилей  
HTML – описывает HTML и BOM

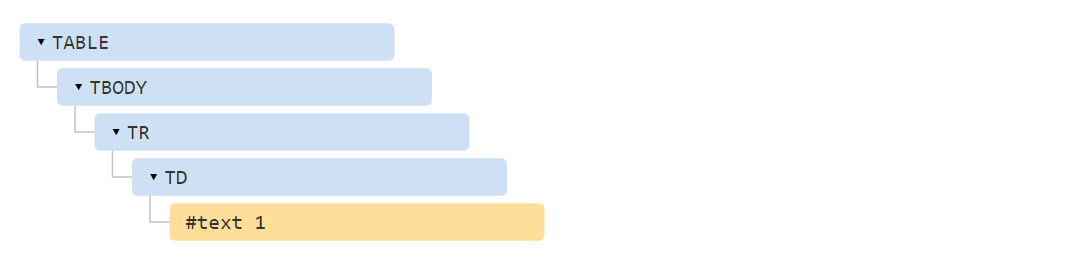
**DOM-дерево**В соответствии с DOM каждый HTML-тег является объектом, как и текст который находится внутри

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  


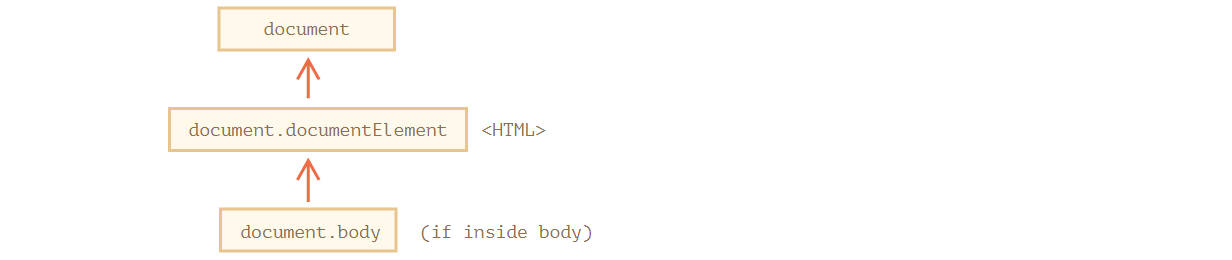
DOM-структура будет такой  


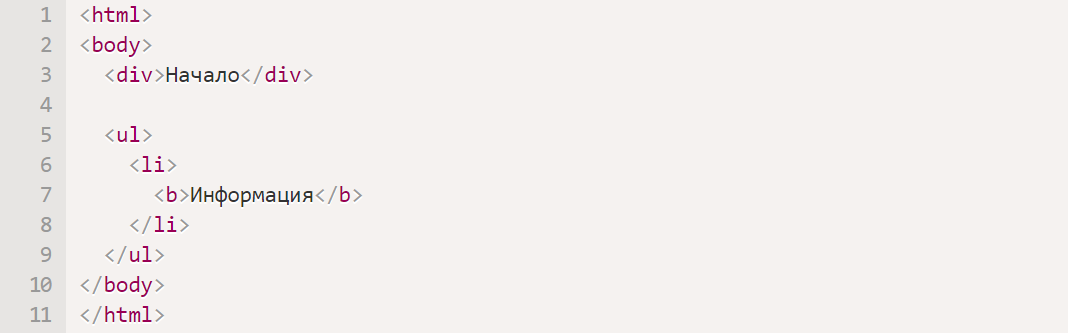
Существуют и другие типы узлов, кроме элементов и текстовых узлов, например комментарий  

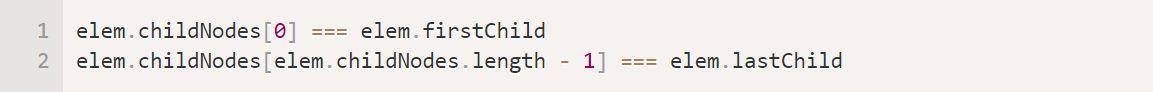

Существует важное правило: если есть что-то в HTML, то оно должно быть и в DOM

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

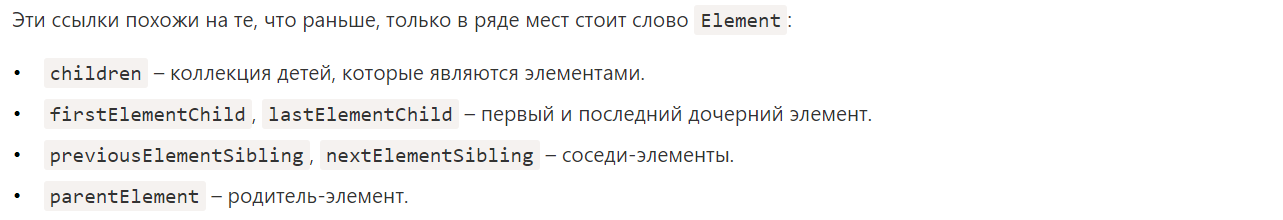
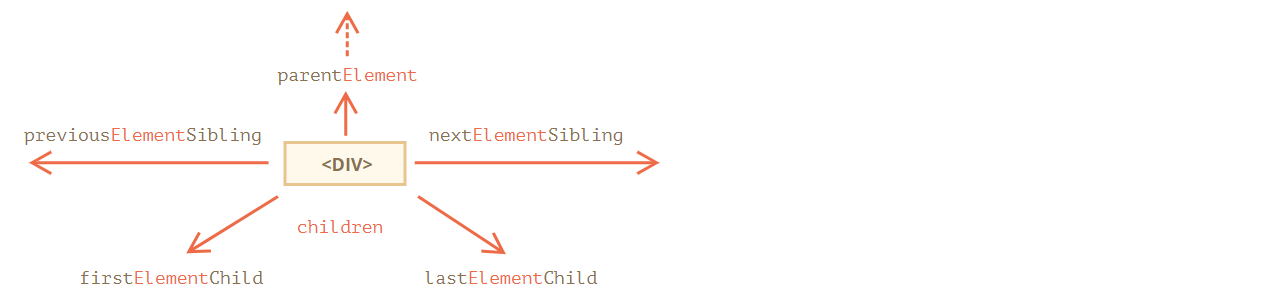
**DOM-коллекции**

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. 



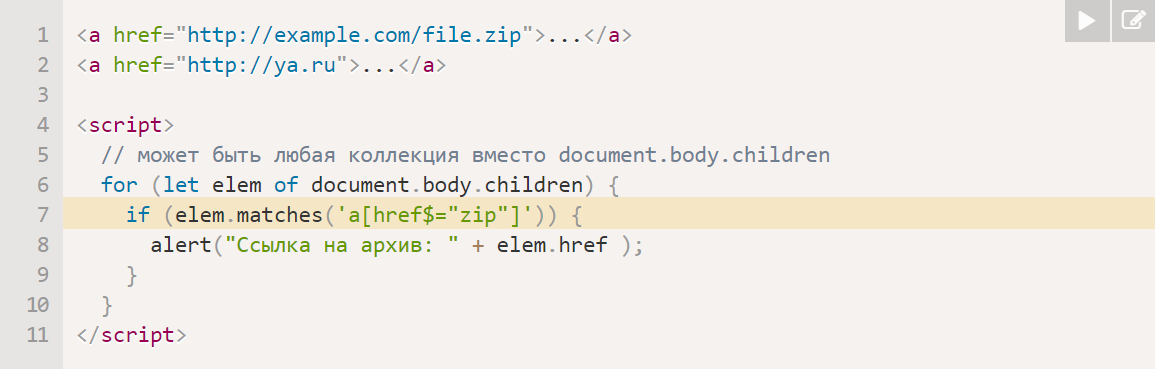
**Поиск по документу  
.getElementById**Значение id должно быть уникальным. Если в документе есть несколько элементов с одинаковым id, то document.getElementById будет непредсказуемым, он будет возвращать любой из этих id случайным образом

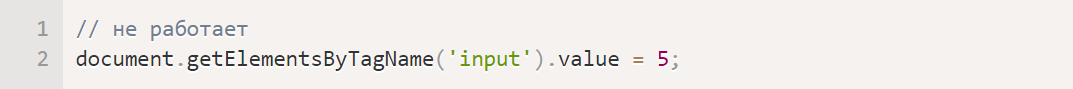
Метод .getElementById можно вызвать только для объекта document, он осуществляет поиск по id внутри документа. Нельзя использовать anyElement.getElementById

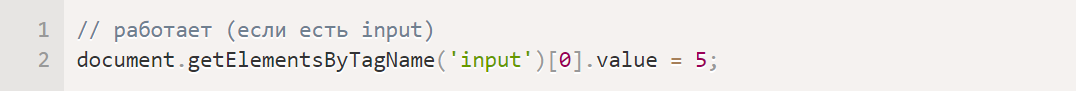
.**querySelectorAll(css) –** самый универсальный метод поиска, который возвращает объект, удовлетворяющий css селектору

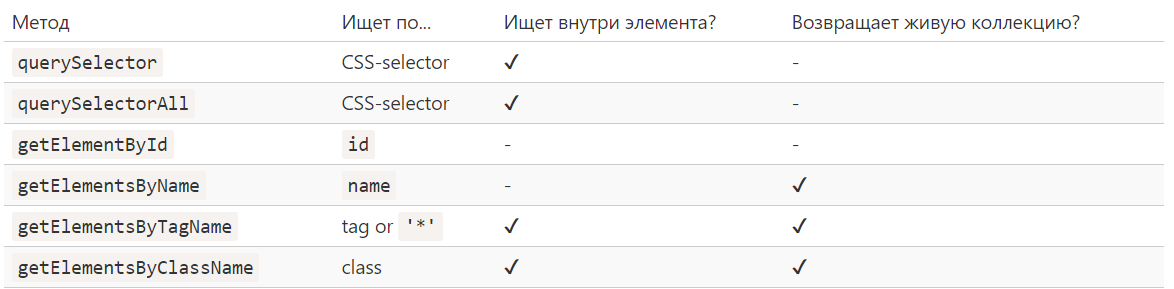
**.querySelector(css)** – работает как .querySelectorAll, но искать и возвращать будет не все элементы, а только первый

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



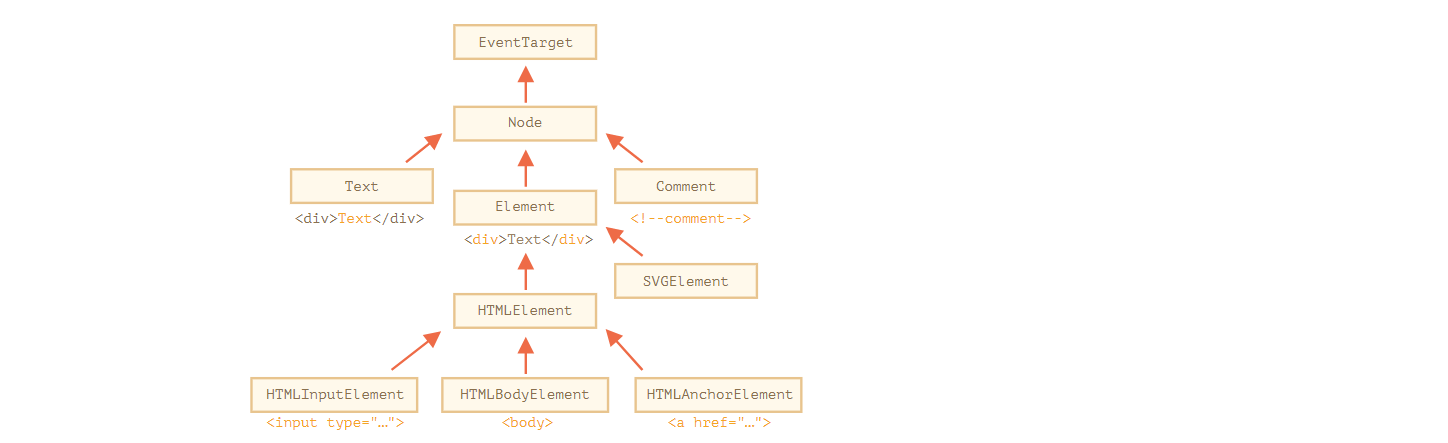
Нельзя менять значение напрямую, так как это коллекция

Работает

Итого

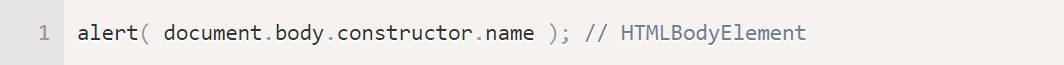
**Свойства узлов: тип, тег и содержимое  
Классы DOM-узлов**У разных DOM-узлов могут быть разные свойства, например у ссылки есть href, у поля ввода есть placeholder, а у кнопки есть type

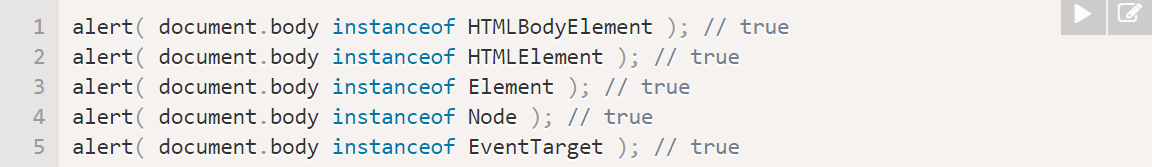
Несмотря на это, у них также есть общие методы и свойства, ведь все классы DOM-узлов образуют единую иерархию

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  


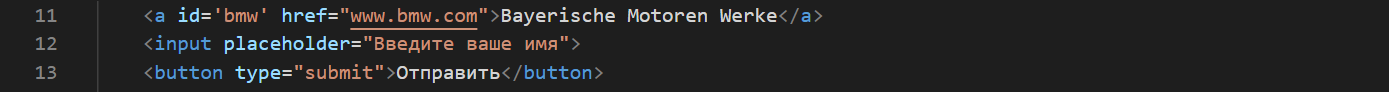
Проверить наследование можно также при помощи instanceOf  


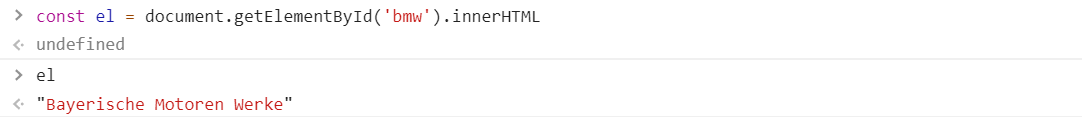
console.log – выводит элемент в виде DOM-дерева  
console.dir – выводит элемент в виде DOM-объекта

**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