**Frontend Questions**

**JS in Browser**

[**Что такое DOM?**](https://youtu.be/1eIRTdgzHtw?t=471)

DOM или Document Object Model (объектная модель документа) — это прикладной программный интерфейс (API) для работы с HTML и XML документами. Когда браузер первый раз читает («парсит») HTML документ, он формирует большой объект, действительно большой объект, основанный на документе — DOM. DOM представляет собой древовидную структуру (дерево документа). DOM используется для взаимодействия и изменения самой структуры DOM или его отдельных элементов и узлов.  
  
Допустим, у нас есть такой HTML:

<!DOCTYPE **html**>

<**html** lang="en">

<**head**>

<**meta** charset="UTF-8">

<**meta** name="viewport" content="width=device-width">

<**meta** http-equiv="X-UA-Compatible" content="ie=edge">

<**title**>Document Object Model</**title**>

</**head**>

<**body**>

<**div**>

<**p**>

<**span**></**span**>

</**p**>

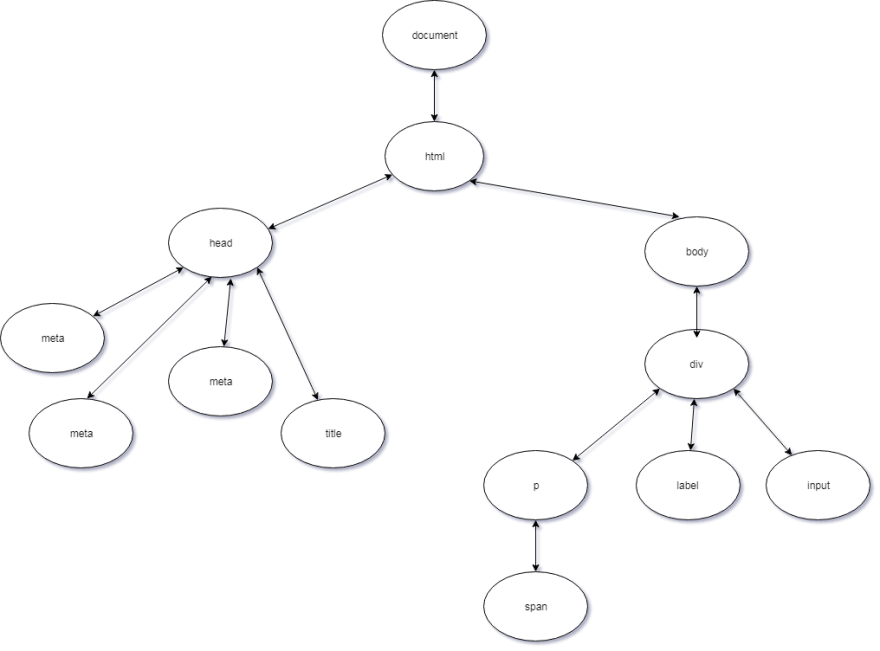
<**label**></**label**>

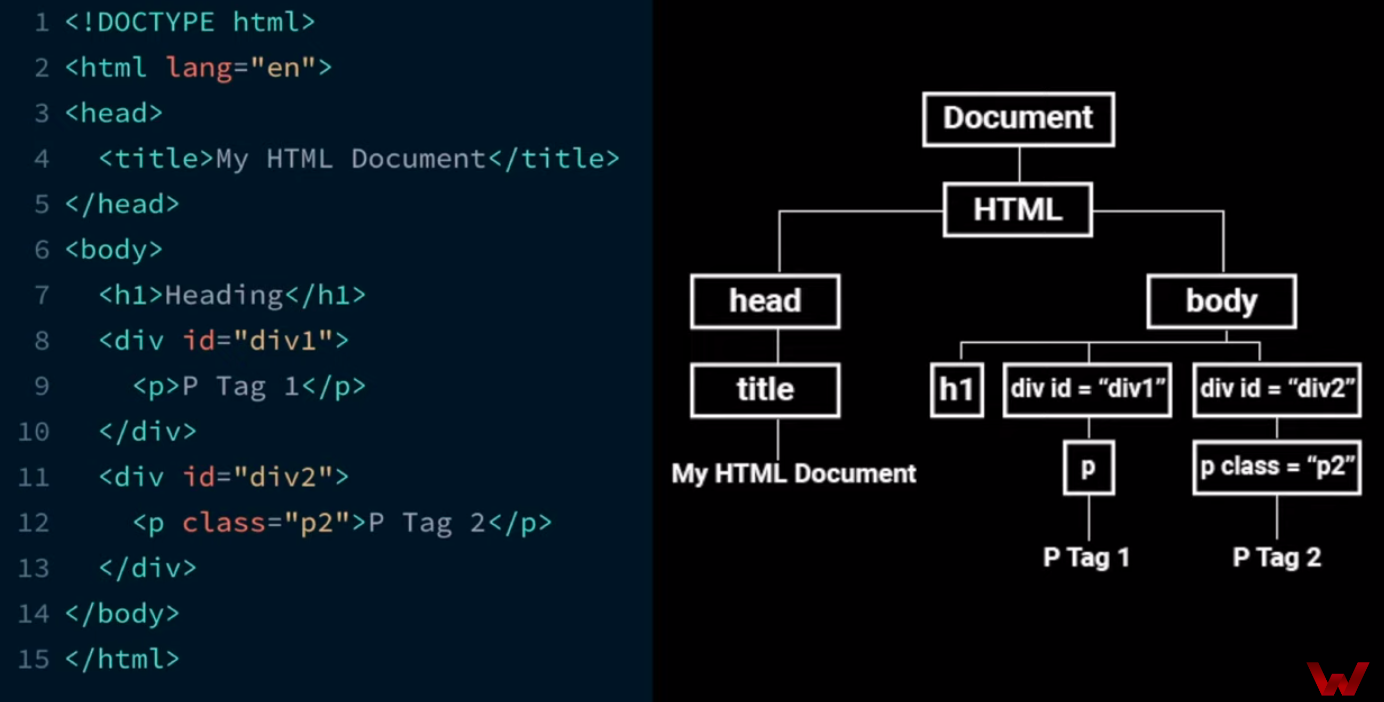
<**input**>

</**div**>

</**body**>

</**html**>

DOM этого HTML выглядит так:  
  
  
  
В JS DOM представлен объектом Document. Объект Document имеет большое количество методов для работы с элементами, их созданием, модификацией, удалением и т.д.



Объектная модель документа, которую браузер создает в памяти компьютера на основании html-кода, полученного им от сервера. Если говорить просто, то html-код – это текст html страницы, а dom – это набор связанных объектов, созданных браузером при парсинге этого текста. По сути, это интерфейс, при помощи которого js может взаимодействовать с элементами страницы. DOM имеет древовидную структуру, по его элементам можно перемещаться, изменять, добавлять события и различные обработчики.

[**Что такое распространение события (Event Propagation)?**](https://youtu.be/1eIRTdgzHtw?t=522)

****

Это механизм, который отрабатывает, когда какое-либо событие происходит в документе. Событие распространяется от объекта window до вызывающего его элемента. При этом событие последовательно затрагивает всех предков целевого документа. Есть три основные фазы:

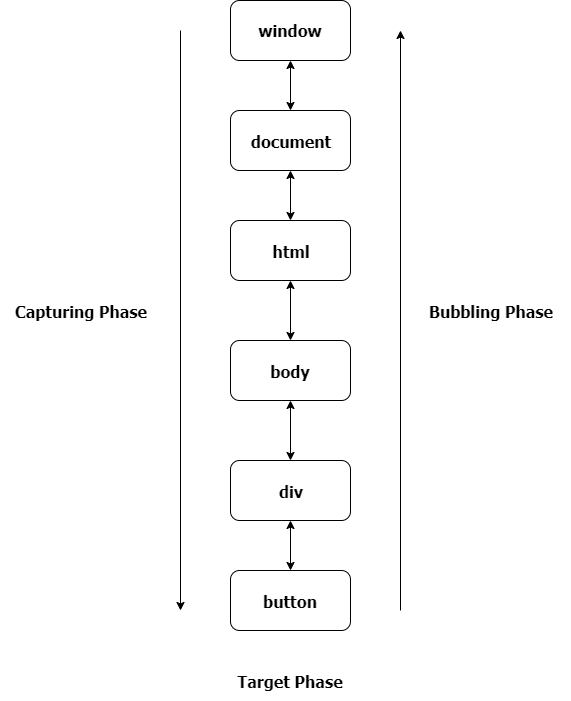
1) capturing – фаза захвата: Событие начинается от корня документа и проходит по dom дереву до целевого элемента.

2) target – событие достигает целевого элемента. Обычно его называют event target.

3) bubbling – фаза всплытия. Событие возвращается обратно до window, параллельно вызывая все события на родительских элементах.

Когда какое-либо событие происходит в элементе DOM, оно на самом деле происходит не только в нем. Событие «распространяется» от объекта Window до вызвавшего его элемента (event.target). При этом событие последовательно пронизывает (затрагивает) всех предков целевого элемента. Распространение события имеет три стадии или фазы:

1. Фаза погружения (захвата, перехвата) — событие возникает в объекте Window и опускается до цели события через всех ее предков.
2. Целевая фаза — это когда событие достигает целевого элемента.
3. Фаза всплытия — событие поднимается от event.target, последовательно проходит через всех его предков и достигает объекта Window.

  
  
Подробнее о распространении событий можно почитать [здесь](https://learn.javascript.ru/bubbling-and-capturing) и [здесь](https://www.w3.org/TR/DOM-Level-3-Events/" \l "event-flow).

#### Что такое всплытие события?

Когда событие происходит в элементе DOM, оно затрагивает не только этот элемент. Событие «всплывает» (подобно пузырьку воздуха в воде), переходит от элемента, вызвавшего событие (event.target), к его родителю, затем поднимается еще выше, к родителю родителя элемента, пока не достигает объекта Window.  
  
Допустим, у нас есть такая разметка:

<**div** class="grandparent">

<**div** class="parent">

<**div** class="child">1</**div**>

</**div**>

</**div**>

И такой JS:

**function** **addEvent**(el, event, callback, isCapture = false) {

**if** (!el || !event || !callback || **typeof** callback !== 'function') **return**

**if** (**typeof** el === 'string') {

el = document.querySelector(el)

}

el.addEventListener(event, callback, isCapture)

}

addEvent(document, 'DOMContentLoaded', () => {

**const** child = document.querySelector('.child')

**const** parent = document.querySelector('.parent')

**const** grandparent = document.querySelector('.grandparent')

addEvent(child, 'click', **function**(e) {

console.log('child')

})

addEvent(parent, 'click', **function**(e) {

console.log('parent')

})

addEvent(grandparent, 'click', **function**(e) {

console.log('grandparent')

})

addEvent('html', 'click', **function**(e) {

console.log('html')

})

addEvent(document, 'click', **function**(e) {

console.log('document')

})

addEvent(window, 'click', **function**(e) {

console.log('window')

})

})

У метода addEventListener есть третий необязательный параметр — useCapture. Когда его значение равняется false (по умолчанию), событие начинается с фазы всплытия. Когда его значение равняется true, событие начинается с фазы погружения (для «прослушивателей» событий, прикрепленных к цели события, событие находится в целевой фазе, а не в фазах погружения или всплытия. События в целевой фазе инициируют все прослушиватели на элементе в том порядке, в котором они были зарегистрированы независимо от параметра useCapture — прим. пер.). Если мы кликнем по элементу child, в консоль будет выведено: child, parent, grandparent, html, document, window. Вот что такое всплытие события.

#### Что такое погружение события?

Когда событие происходит в элементе DOM, оно происходит не только в нем. В фазе погружения событие опускается от объекта Window до цели события через всех его предков.  
  
Разметка:

<**div** class="grandparent">

<**div** class="parent">

<**div** class="child">1</**div**>

</**div**>

</**div**>

JS:

**function** **addEvent**(el, event, callback, isCapture = false) {

**if** (!el || !event || !callback || **typeof** callback !== 'function') **return**

**if** (**typeof** el === 'string') {

el = document.querySelector(el);

}

el.addEventListener(event, callback, isCapture)

}

addEvent(document, 'DOMContentLoaded', () => {

**const** child = document.querySelector('.child')

**const** parent = document.querySelector('.parent')

**const** grandparent = document.querySelector('.grandparent')

addEvent(child, 'click', **function**(e) {

console.log('child');

}, true)

addEvent(parent, 'click', **function**(e) {

console.log('parent')

}, true)

addEvent(grandparent, 'click', **function**(e) {

console.log('grandparent')

}, true)

addEvent('html', 'click', **function**(e) {

console.log('html')

}, true)

addEvent(document, 'click', **function**(e) {

console.log('document')

}, true)

addEvent(window, 'click', **function**(e) {

console.log('window')

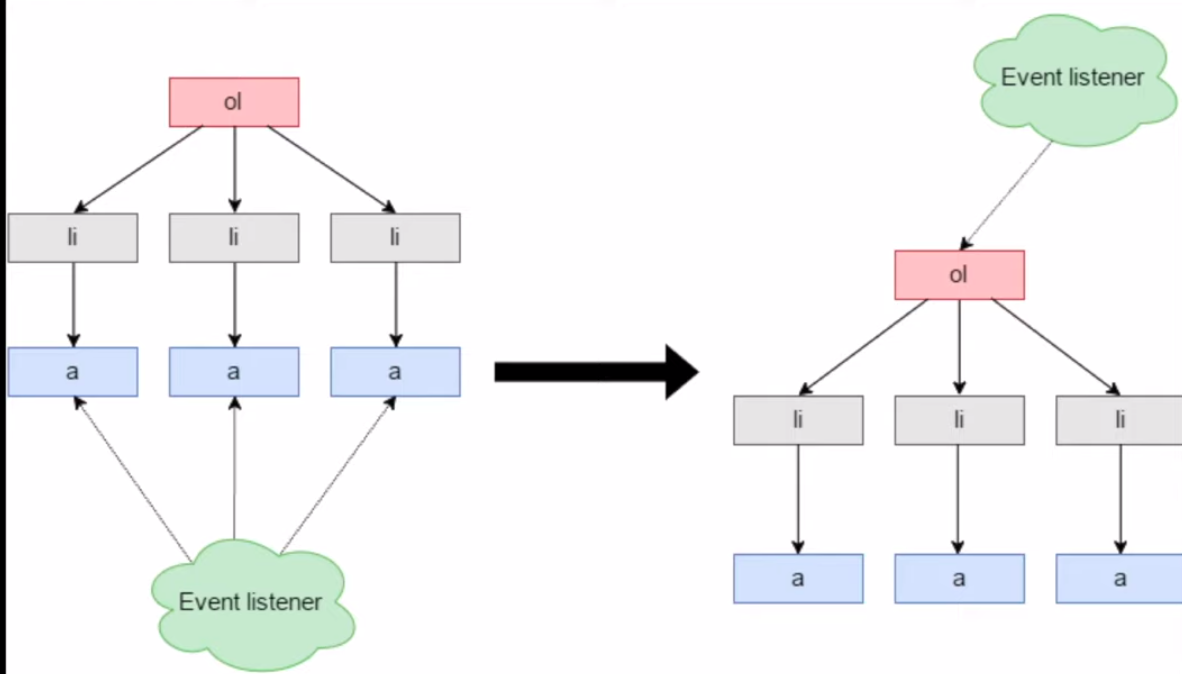
}, true)

})

У метода addEventListener есть третий необязательный параметр — useCapture. Когда его значение равняется false (по умолчанию), событие начинается с фазы всплытия. Когда его значение равняется true, событие начинается с фазы погружения. Если мы кликнем по элементу child, то увидим в консоли следующее: window, document, html, grandparent, parent, child. Это и есть погружение события.

<https://learn.javascript.ru/bubbling-and-capturing>

[**Что такое делегирование событий (Event Delegation)?**](https://youtu.be/1eIRTdgzHtw?t=576)



Делегирование событий - это приём, заключающийся в добавлении обработчиков событий к родительскому элементу, а не к дочерним элементам. Обработчик будет срабатывать всякий раз, когда событие будет запущено на дочерних элементах благодаря всплытию событий в DOM. Элемент, на котором непосредственно произошло событие, можно отслеживать при помощи event target.

https://learn.javascript.ru/event-delegation

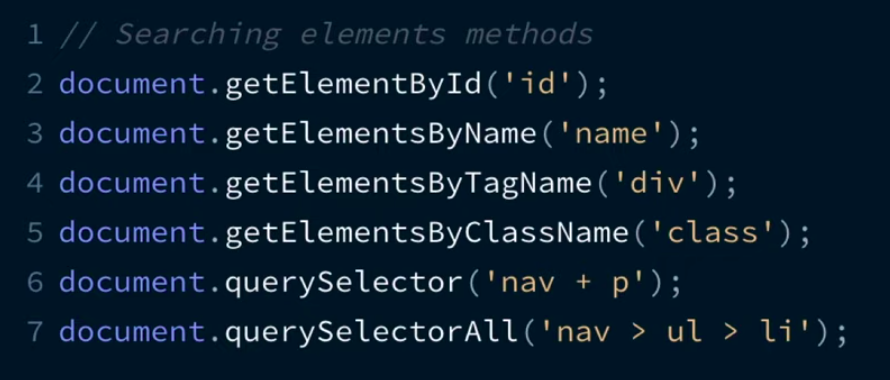
[**Разница между e.preventDefault() и e.stopPropagation()?**](https://youtu.be/CjdCxxqObaM?t=650)

Метод event.preventDefault() отключает поведение элемента по умолчанию. Если использовать этот метод в элементе form, то он предотвратит отправку формы (submit). Если использовать его в contextmenu, то контекстное меню будет отключено (данный метод часто используется в keydown для переопределения клавиатуры, например, при создании музыкального/видео плеера или текстового редактора — прим. пер.). Метод event.stopPropagation() отключает распространение события (его всплытие или погружение).



#### Как узнать об использовании метода event.preventDefault()? Для этого мы можем использовать свойство event.defaulPrevented, возвращающее логическое значение, служащее индикатором применения к элементу метода event.preventDefault.

[**Методы поиска элементов в DOM?**](https://youtu.be/CjdCxxqObaM?t=678)

****

https://learn.javascript.ru/searching-elements-dom

Eсли у элемента есть атрибут id, то мы можем получить его вызовом document.**getElementById(id)**, где бы он ни находился.

elem.**getElementsByTagName(tag)** ищет элементы с данным тегом и возвращает их коллекцию. Передав "\*" вместо тега, можно получить всех потомков.

elem.**getElementsByClassName(className)** возвращает элементы, которые имеют данный CSS-класс.

document.**getElementsByName(name)** возвращает элементы с заданным атрибутом name. Очень редко используется.

#### Метод elem.querySelector(css) возвращает первый элемент, соответствующий данному CSS-селектору.

#### Самый универсальный метод поиска – это elem.querySelectorAll(css), он возвращает все элементы внутри elem, удовлетворяющие данному CSS-селектору.

[**Разница между event.target и event.currentTarget?**](https://youtu.be/kx3dR6ztICU?t=539)

#### event. target – самый глубокий элемент, на котором произошло событие, который вызвал событие. event. currentTarget (= this ) – элемент, на котором в данный момент сработал обработчик (до которого «доплыло» событие) либо к которому был прикреплен прослушиватель события.

#### [Разница между .stopPropagation() и .stopImmediatePropagation()?](https://youtu.be/kx3dR6ztICU?t=580)

StopPropagation() отменяет дальнейшее всплытие, но на текущем элементе все обработчики срабатывают. А stopImmediatePropagation не только остановит дальнейшее всплытие, но и остановит обработку события на текущем элементе.

[**Разница между событиями load и DOMContentLoaded?**](https://youtu.be/kx3dR6ztICU?t=627)

DOMContentLoaded — событие происходит, когда браузер разобрал HTML-страницу и составил DOM-дерево; load — событие происходит, когда загрузилась и HTML страница, и все ресурсы для её отображения пользователю: стили, картинки и так далее.

[Подробнее](https://learn.javascript.ru/onload-ondomcontentloaded)

[**Разница между attribute и property у DOM-элементов?**](https://youtu.be/IooJ3P2VUYs?t=659)

Аттрибут – это статичное значение дом-элемента, которое неизменяемо и в большинстве своем может быть добавлено в hrml разметке. А вот свойство – это вычисленное значение дом-элемента. При этом его особенность в том, что оно может динамически изменяться.



https://learn.javascript.ru/attributes-and-custom-properties

[**Разница между HTMLCollection и NodeList?**](https://youtu.be/IooJ3P2VUYs?t=705)

[**HTMLCollection**](https://youtu.be/IooJ3P2VUYs?t=705) **–** это динамическая коллекция, которая представляет собой массивоподобный и итерируемый объект дочерних элементов. NodeList – статический список нод и узлов, в который входят все найденные в документе элементы,

[Подробнее](https://doka.guide/js/htmlcollection-and-nodelist/)

[**Как динамически добавить элемент на HTML-страницу?**](https://youtu.be/nvktMVFM0_M?t=551)

<https://learn.javascript.ru/modifying-document>

[**Типы узлов DOM-дерева?**](https://youtu.be/7TvS0iKR3_c?t=201)

Существует 12 типов узлов. Но на практике мы в основном работаем с 4 из них:

* document – «входная точка» в DOM.
* узлы-элементы – HTML-теги, основные строительные блоки.
* текстовые узлы – содержат текст.
* комментарии – иногда в них можно включить информацию, которая не будет показана, но доступна в DOM для чтения JS.

https://learn.javascript.ru/dom-nodes

https://habr.com/ru/companies/ruvds/articles/539096/

**Свойства для перемещения по DOM-дереву?**

<https://learn.javascript.ru/dom-navigation>

<https://www.w3schools.com/js/js_htmldom_navigation.asp>

**Виды событий в JavaScript?**

Все события JavaScript можно разделить на следующие категории: события мыши (Mouse Events), события клавиатуры (Keyboard Events), события объектов и фреймов (Frame/Object Events), события формы и элементов управления (Form Events), события перетаскивания (Drag Events), события анимации (Animation Events), события буфера обмена (Clipboard Events), события мультимедиа (Media Events), события трансформации (Transition Events), события, посылаемые сервером (Server-Sent Events), события касания (Touch Events), события печати (Print Events), разные события (Misc Events).

[Подробнее](https://learn.javascript.ru/introduction-browser-events)

[**Как добавить обработчик события на DOM-элемент?**](https://youtu.be/7TvS0iKR3_c?t=425)

[Подробнее](https://learn.javascript.ru/introduction-browser-events)

https://www.w3schools.com/js/js\_htmldom\_events.asp

[**Как удалить обработчик события с DOM-элемента?**](https://youtu.be/7TvS0iKR3_c?t=505)

removeEventListener() Удаляет обработчик события, который был зарегистрирован при помощи EventTarget.

[**Сколько аргументов принимает addEventListener?**](https://youtu.be/7TvS0iKR3_c?t=538)

https://www.w3schools.com/jsref/met\_element\_addeventlistener.asp

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

Данный метод принимает три аргумента: тип события, исполняемую функцию и булево значение (значение логического типа). Первый атрибут — это имя события. Второй — функция, которая будет исполняться при наступлении события. Это может быть ссылка на функцию (как в нашем случае) или просто анонимная функция. Третий атрибут, принимающий значение true или false, указывает, каким образом срабатывают множественные события. Например, мы прослушиваем событие click на двух вложенных элементах (один элемент вложен в другой). Когда пользователь щелкает на этих элементах, два события click срабатывают в порядке, определяемом этим атрибутом. Если его значение для одного из элементов равно true, то считается, что событие для этого элемента произошло первым, а для другого элемента — вторым по счету. Обычно для большинства ситуаций хватает значения false.

[**Разница между innerHTML и outerHTML?**](https://youtu.be/xZLxdts7ZW4?t=621)

Свойство outerHTML содержит HTML элемента целиком. Это как innerHTML плюс сам элемент. Будьте осторожны: в отличие от innerHTML , запись в outerHTML не изменяет элемент. Вместо этого элемент заменяется целиком во внешнем контексте.

https://www.makeuseof.com/html-inner-outer-dom-difference/

The innerHTML property is part of the DOM and you can access it via JavaScript. You can use it to get or set the contents of an element. This content excludes the element's own tag and only includes the markup that represents the element's children, in the form of a string.

The outerHTML property is like innerHTML in many ways. You can use it to get or set the contents of a selected element. However, it also sets the markup representing the element itself. This includes the opening tag, any properties, and—where relevant—the closing tag.

The innerHTML and outerHTML DOM properties have many similarities, but one key difference. The innerHTML property captures the HTML contents of an element. In contrast, the outerHTML property captures the HTML that represents the element itself and its content.

You can use these properties to read and write HTML content via the DOM. The DOM will be a common, important concept throughout your JavaScript development process.

[**Расскажите про координаты в браузере?**](https://youtu.be/70VnuTXi4Wk?t=644)

Система координат в браузере аналогична плоской системе координат из школьной алгебры. Есть две оси: ось y и ось x, на этой плоскости располагаются все элементы сайта. Единственным отличием от классической системы координат является то, что положительное направление оси y в браузере повёрнуто вниз.

<https://learn.javascript.ru/coordinates> [Дока](https://doka.guide/js/coordinates/" \l ":~:text=Система координат в браузере аналогична,y в браузере повёрнуто вниз.)

[**Разница между JSON и XML?**](https://youtu.be/XtQPrt8G0n8?t=28)

Формат JSON используется для хранения и передачи данных, а XML - для представления данных в машиночитаемом виде. JSON набирает популярность как средство хранения данных для веб-приложений благодаря своей простоте. В отличие от него, XML все еще используется для передачи структурированных данных через Интернет.

https://coderlessons.com/tutorials/veb-razrabotka/arkhitektura-veb-servisov/10-json-protiv-xml

<https://www.w3schools.com/js/js_json_xml.asp>

[**Для чего используется свойство window.navigator?**](https://youtu.be/lZNWrW39ELM?t=493)

Свойство window.navigator возвращает объект описания приложения (user agent), которое выполняет скрипт. В подавляющем большинстве случаев это приложение — браузер. Объект содержит свойства, описывающие браузер, и методы для выполнения действий.

Часто используемые свойства:

userAgent возвращает строку, которая содержит название браузера. Не стоит использовать это свойство, чтобы определить браузер пользователя! Спецификация рекомендует браузерам передавать минимум информации в userAgent, значение может меняться от версии к версии.

language возвращает предпочитаемый язык интерфейса в виде языкового тега. Например, en, ru, en-US и т.д. Обычно это язык, установленный в настройках браузера.

languages возвращает массив предпочитаемых языков в порядке предпочтительности. Первый в списке будет язык, который возвращает navigator.language.

cookieEnabled возвращает true, если браузер пользователя поддерживает куки и они включены, false в противном случае.

onLine возвращает true, если у пользователя есть подключение к сети. Браузеры вкладывают разные смыслы в понятие «онлайн», поэтому это свойство — ненадёжный источник данных.

Объект navigator содержит множество других свойств, большинство из них экспериментальные или поддерживаются конкретными браузерами.

Методы объекта navigator служат для взаимодействия с другими WebAPI. Например, метод vibrate, который вызывает вибрацию пользовательского устройства, если она поддерживается: navigator.vibrate(200)

[**Для чего используется метод .focus()?**](https://youtu.be/lZNWrW39ELM?t=551)

Метод focus устанавливает фокус на элементе (чаще всего на инпуте). Это значит, что в этом инпуте начнет моргать курсор и вводимый с клавиатуры текст будет попадать именно в этот инпут.

[ДОКА](https://doka.guide/js/element-focus/)

[**Для чего используется свойство .forms?**](https://youtu.be/lZNWrW39ELM?t=616)

Свойство Form используется для ссылки на форму или для ссылки на форму, связанную с вложенным элементом управления.

https://learn.javascript.ru/form-elements

https://www.w3schools.com/js/js\_validation.asp

[**Для чего используется метод .scrollIntoView()?**](https://youtu.be/lZNWrW39ELM?t=682)

Метод scrollIntoView() позволяет программно прокрутить окно до определённого элемента.

[ДОКА](https://doka.guide/js/element-scrollintoview/" \l ":~:text=Метод scrollIntoView() позволяет программно прокрутить окно до определённого элемента.)

[**Как использовать media выражения в JavaScript?**](https://youtu.be/3kvKFfPteFg?t=30)

[Ответ](https://webdevblog.ru/rabotaem-s-media-zaprosami-cherez-javascript/)

[**Разница между методами .submit() и .requestSubmit()?**](https://youtu.be/3kvKFfPteFg?t=90)

As you can see the name input has a required attribute, the form will however still be submitted, because submit() doesn't care about validation.

This is where requestSubmit() is useful. It acts exactly like a trigger on a submit button. If you use form.requestSubmit() instead of form.submit() the form will only be submitted if it is valid. If you leave the name input empty the browser will show a hint to the user that they have to fill out that field and the form will not be submitted.

Browser support is pretty good, except for Internet Explorer and more importantly Safari. So, you should make a feature detection before using it.

**Расскажите о IntersectionObserver?**

Intersection Observer — браузерный API, который позволяет асинхронно отслеживать пересечение элемента с его родителем или областью видимости документа (viewport). В момент пересечения можно запустить какое-либо действие, например, подгрузить дополнительные посты в ленте новостей («бесконечный скролл») или сделать «ленивую» загрузку контента.

[ДОКА](https://doka.guide/js/intersection-observer/)

[**Расскажите о URLSearchParams?**](https://youtu.be/hL5yFo9Pms4?t=98)

URLSearchParams — это класс, предоставляющий удобное API для формирования строки поисковых параметров, которую потом можно использовать для формирования полного адреса. Все параметры в строке будут закодированы для безопасной вставки в адрес. Также этот класс можно встретить как часть класса URL.

[ДОКА](https://doka.guide/js/urlsearchparams/" \l ":~:text=URLSearchParams — это класс%2C предоставляющий удобное,встретить как часть класса URL .)

https://developer.mozilla.org/ru/docs/Web/API/URLSearchParams

[**Какие есть ограничения у window.close()?**](https://youtu.be/hL5yFo9Pms4?t=149)

[Ответ](https://habr.com/ru/companies/ruvds/articles/550194/)

### [Async JS](https://youtu.be/kx3dR6ztICU?t=681)

**Разница между синхронными и асинхронными функциями?**

Синхронные функции являются блокирующими, в то время как асинхронные нет. Другими словами, во время выполнения js, когда интерпретатор наталкиваеся на синхронную функцию, он блокирует дальнейшее выполнение операций, пока данная функция не будет выполнена. Поэтому набор таких функций выполняется последовательно одна за другой. Асинхронные функции не блокируют дальнейшее выполнение скрипта. Именно поэтому различные тяжелые операции типа запроса данных делают асинхронными. Обычно такие функции в качестве аргумента принимают коллбек – это еще одна функция, которая вызовется, как только выполнится асинхронная, и которая сможет обработать полученный результат.



<https://stasonmars.ru/javascript/polnoe-ponimanie-syncronnogo-i-asyncronnogo-javascript-s-async-await/>

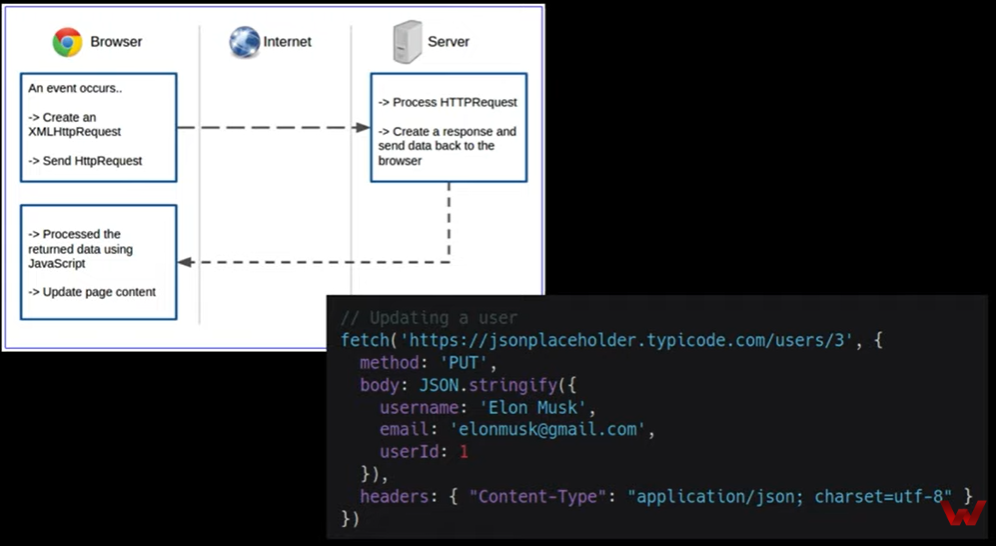
В синхронном коде выполнение функций происходит в том же месте, где они были вызваны, и в тот момент, когда происходит вызов. В асинхронном коде всё по-другому. Вызов функции не означает, что она отработает прямо здесь и сейчас. Более того, мы не знаем, когда она отработает.

[Еще информация](https://ru.hexlet.io/courses/js-asynchronous-programming/lessons/asynchronous-code/theory_unit" \l ":~:text=В синхронном коде выполнение функций,отработает прямо здесь и сейчас.)

<https://www.w3schools.com/js/js_callback.asp>

**Что такое AJAX?**

Для сетевых запросов из JavaScript есть широко известный термин «AJAX» (аббревиатура от Asynchronous JavaScript And XML). Это не технология, а термин, который описывает подход использования нескольких существующих технологий вместе для работы на стороне клиента и для создания асинхронных веб-приложений. С помощью AJAX веб-приложения могут отправлять данные на сервер и получать их с сервера асинхронно. Таким образом происходит отделение логики представления, то есть, отрисовки UI, от логики обмена данными. В результате страницы могут динамически изменять содержимое без полной перезагрузки, поэтому они работают быстрее и становятся более отзывчивыми к действиям пользователя. На практике же, для получения и передачи данных используют формат данных JSON вместо XML из-за того, что JSON основан на JS. А для работы с асинхронными запросами используют новый метод fetch вместо устаревшего xmlHttpRequest.



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

[То, чем AJAX был раньше](https://www.w3schools.com/js/js_ajax_intro.asp)

AJAX или Asyncronous JavaScript and XML — это набор взаимосвязанных технологий, которые позволяют работать с данными в асинхронном режиме. Это означает, что мы можем отправлять данные на сервер и получать данные с него без перезагрузки веб-страницы.

AJAX использует следующие технологии:

HTML — структура веб-страницы.

CSS — стили веб-страницы.

JavaScript — поведение страницы и работа с DOM.

XMLHttpRequest API — отправка и получение данных с сервера.

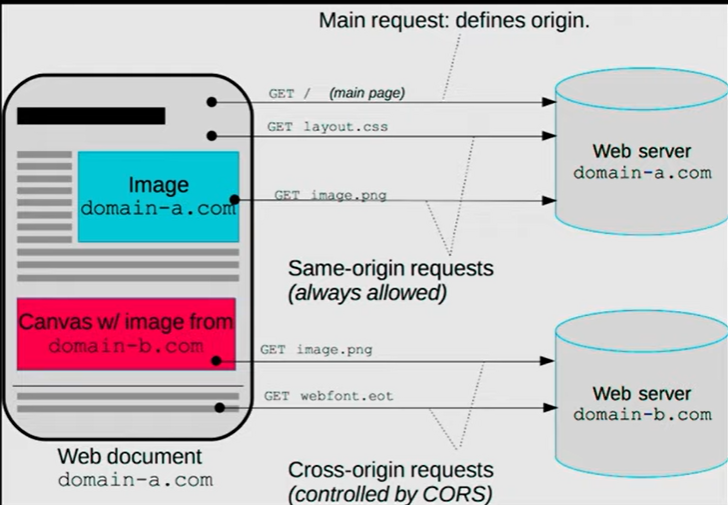
PHP, Python, Nodejs — какой-нибудь серверный язык.

**Что такое same-origin policy в контексте JavaScript?**

same-origin policy или принцип одинакового источника определяет, как документ или скрипт, загруженный из одного источника, может взаимодействовать с ресурсом из другого источника. Другими словами, этот принцип не позволяет js выполнять запросы за границы домена. Источник определяется как комбинация схемы URI, имени хоста и номера порта. Это помогает изолировать потенциально вредоносные документы и не дает вредоносному сценарию на одной странице получить доступ к конфиденциальным данным на другой. А для того, чтобы разрешить такие кроссдоменные запросы, используется cors.

[Подробнее](https://developer.mozilla.org/ru/docs/Web/Security/Same-origin_policy)

<https://learn.javascript.ru/cross-window-communication>



**Что такое цикл событий (event loop) и как он работает?**

В JavaScript асинхронность — основной инструмент, который обрабатывает запросы параллельно с загрузкой веб-страницы. Сейчас невозможно представить интернет, где все запросы на сервер отправлялись бы с перезагрузкой страницы.

Любые данные от сервера запрашиваются асинхронно: отправляется запрос (XMLHttpRequest или XHR), и код📟 не ждёт его возвращения🔄, продолжая выполняться. Когда же сервер отвечает, объект XHR получает уведомление об этом и запускает функцию⚙️ обратного вызова — callback, который передали в него перед отправкой запроса.

Если правильно использовать инструменты языка👅, то выполнение запроса, который происходит последовательно и в одном потоке, никак не мешает приёму событий и реакции на них — человек👨 спокойно работает с интерфейсом, не замечая лагов, сбоев и зависаний.

Event loop в JavaScript — менеджер асинхронных вызовов.

Чтобы этот хитрый процесс слаженно работал, в JavaScript реализован механизм для управления очерёдностью исполнения кода📟 . Поскольку это однопоточный язык👅, возникла необходимость "вклиниваться" в текущий контекст исполнения. Этот механизм называется event loop — событийный цикл.

С английского loop переводится как "петля", что отлично отражает смысл: мы имеем дело с закольцованной очередью.

Event loop регулирует последовательность исполнения контекстов — стек. Он формируется, когда сработало событие или была вызвана функция⚙️. Реакция на событие помещается в очередь исполнения, в event loop, который последовательно, с каждым циклом выполняет попадающий в него код📟 . При этом привязанная к событию функция⚙️ вызывается следующей после текущего контекста исполнения.

В JavaScript постоянно работают связанные между собой синхронная и асинхронная очереди выполнения. Синхронная — stack — формирует очередь и пробрасывает в асинхронную — event loop — вызовы функций⚙️, которые будут выполнены после текущего запланированного исполняемого контекста.

Чтобы данные находились в консистентном состоянии, каждая функция⚙️ должна быть выполнена до конца. Это обусловлено однопоточностью JavaScript и некоторыми другими особенностями, например характерными для функциональных ⚙️языков👅 программирования замыканиями. Поэтому единственный поток представлен в виде очереди контекстов исполнения, в которой и происходит "вклинивание" функций⚙️, прошедших через цикл событий.

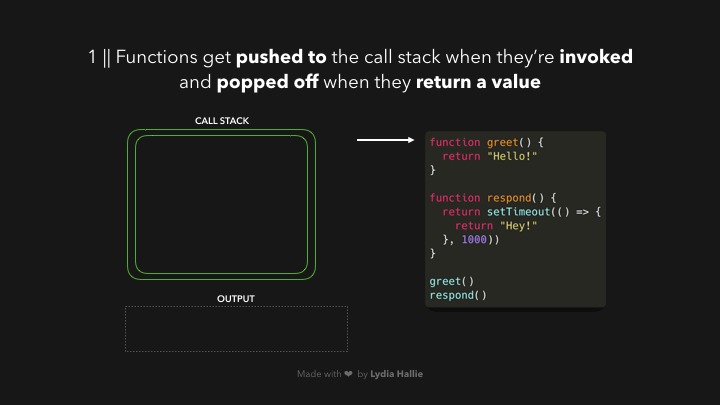
**Описание[​](https://www.jscamp.app/ru/docs/javascript25/" \l "описание)**

JavaScript это однопоточный язык: одновременно может выполняться только одна задача. Обычно в этом нет ничего сложного, но теперь представьте, что вы запускаете задачу, которая занимает 30 секунд... Да. Во время этой задачи мы ждем 30 секунд, прежде чем что-либо еще может произойти (по умолчанию JavaScript запускается в главном потоке браузера, поэтому весь пользовательский интерфейс будет ждать)😬 Сейчас 2021 год, никто не хочет медленный сайт который тупит.

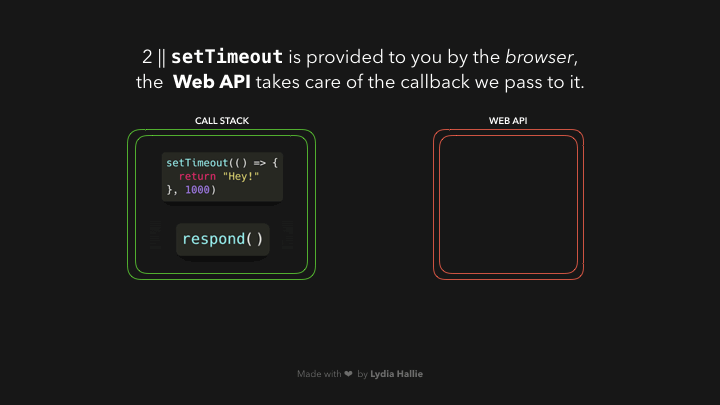
К счастью, браузер предоставляет нам некоторые функции, которые сам механизм JavaScript не предоставляет: Web API. Который включает в себя DOM API, setTimeout, HTTP-запросы и так далее. Это может помочь нам создать асинхронное неблокирующее поведение 🚀.

Когда мы вызываем функцию, она добавляется в call stack(стек вызовов). Стек вызовов является частью механизма JS, это не зависит от браузера. Это классический взгляд на стек, т.е first in, last out. Когда функция возвращает значение, она "выталкивается" из стека.

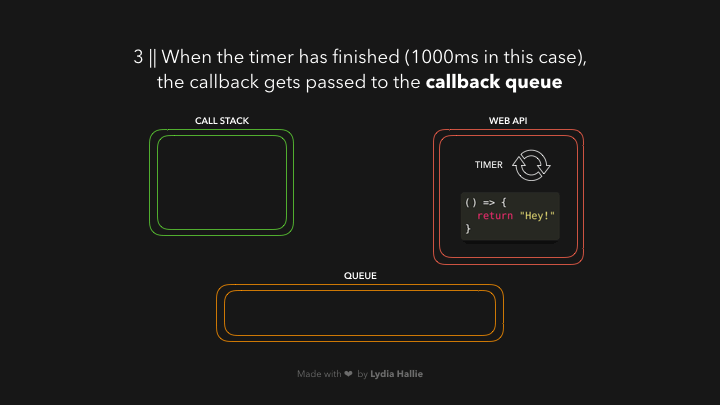
*function* great() {  
 *return* 'Hello'  
}  
  
*function* respond() {  
 *return* setTimeout(() => alert('Hey!'), 1000)  
}  
  
great()  
respond()



Функция respond возвращает функцию setTimeout. SetTimeout предоставляется нам через Web-API: он позволяет нам делить задачи, не блокируя основной поток. Callback функция, которую мы передали в функцию setTimeout, лямбда функция () => {return 'Hey'} добавляется в Web-API. Тем временем setTimeout и responde извлекаются из стека и возвращают свои значения.

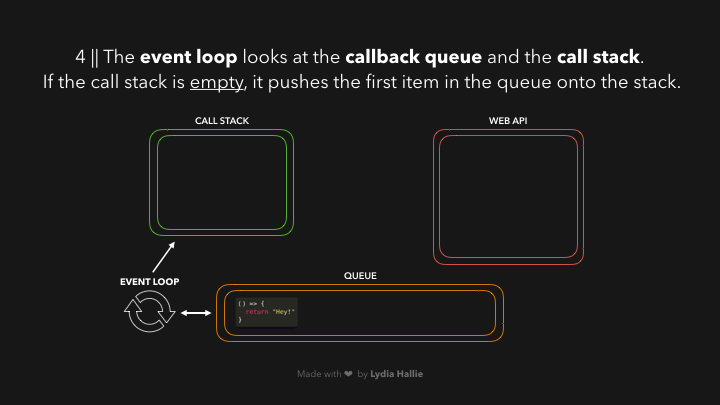


В Web-API таймер работает до тех пор, пока второй аргумент, который мы передали ему, не подождет 1000 мс. Callback не сразу добавляется в стек вызовов, а передается в нечто, называемое очередью.



Это может сбивать с толку: это не означает, что callback функция добавляется в стек вызовов (таким образом, возвращает значение) через 1000 мс! Он просто добавляется в очередь через 1000 мс. Но в этой очереди, функция должна ждать пока придет ее черёд.

Теперь это та часть, которую мы все ждали... Время для event loop выполнить единственную задачу: соединить очередь со стеком вызовов! Если стек вызовов пуст, то есть, если все ранее вызванные функции вернули свои значения и были извлечены из стека, первый элемент в очереди добавляется в стек вызовов. В этом случае никакие другие функции не были вызваны, что означает, что стек вызовов был пуст к тому времени, когда callback функция была первым элементом в очереди.



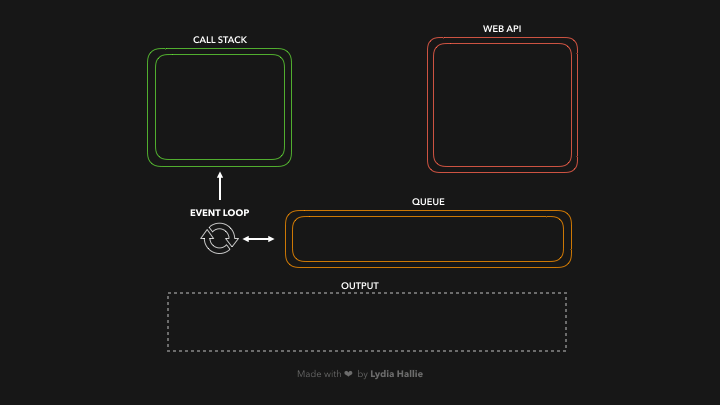
callback добавляется в стек вызовов, вызывается и возвращает значение, а также извлекается из стека.



Смотреть весело, но вы не сможете полностью понять тему, не работая с ней снова и снова. Попробуйте выяснить, что появится в консоли, если мы запустим следующее:

*const* foo = () => console.log('First')  
*const* bar = () => setTimeout(() => console.log('Second'), 500)  
*const* baz = () => console.log('Third')  
  
bar()  
foo()  
baz()

Давайте посмотрим, что происходит, когда мы запускаем этот код в браузере:



Мы вызываем bar, которая возвращает функцию setTimeout. Callback который мы передали в setTimeout добавляется в Web API, функция setTimeout и bar извлекаются из стека вызовов.

Таймер запускается, тем временем foo вызывается и записывает в журнал First. foo возвращает undefined, baz вызывается и callback добавляется в очередь baz логирует Third. Цикл обработки событий видит, что коллстек пуст после возврата baz, после чего колбэк добавляется в стек вызовов. Callback логирует Second.

<https://learn.javascript.ru/event-loop>

<https://medium.com/devschacht/javascript-eventloop-explained-f2dcf84e36ee>

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

<https://tproger.ru/blogs/effective-code-in-js-with-event-loop/>

<https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API>

Идея *событийного цикла* очень проста. Есть бесконечный цикл, в котором движок JavaScript ожидает задачи, исполняет их и снова ожидает появления новых.

Общий алгоритм движка:

1. Пока есть задачи:
   * выполнить их, начиная с самой старой
2. Бездействовать до появления новой задачи, а затем перейти к пункту 1

Это формализация того, что мы наблюдаем, просматривая веб-страницу. Движок JavaScript большую часть времени ничего не делает и работает, только если требуется исполнить скрипт/обработчик или обработать событие.

Примеры задач:

* Когда загружается внешний скрипт <script src="...">, то задача – это выполнение этого скрипта.
* Когда пользователь двигает мышь, задача – сгенерировать событие mousemove и выполнить его обработчики.
* Когда истечёт таймер, установленный с помощью setTimeout(func, ...), задача – это выполнение функции func
* И так далее.

Задачи поступают на выполнение – движок выполняет их – затем ожидает новые задачи (во время ожидания практически не нагружая процессор компьютера)

Может так случиться, что задача поступает, когда движок занят чем-то другим, тогда она ставится в очередь.

Очередь, которую формируют такие задачи, называют «очередью макрозадач» (macrotask queue, термин v8).

Например, когда движок занят выполнением скрипта, пользователь может передвинуть мышь, тем самым вызвав появление события mousemove, или может истечь таймер, установленный setTimeout, и т.п. Эти задачи формируют очередь, как показано на иллюстрации выше.

Задачи из очереди исполняются по правилу «первым пришёл – первым ушёл». Когда браузер заканчивает выполнение скрипта, он обрабатывает событие mousemove, затем выполняет обработчик, заданный setTimeout, и так далее.

Пока что всё просто, не правда ли?

Отметим две детали:

1. Рендеринг (отрисовка страницы) никогда не происходит во время выполнения задачи движком. Не имеет значения, сколь долго выполняется задача. Изменения в DOM отрисовываются только после того, как задача выполнена.
2. Если задача выполняется очень долго, то браузер не может выполнять другие задачи, обрабатывать пользовательские события, поэтому спустя некоторое время браузер предлагает «убить» долго выполняющуюся задачу. Такое возможно, когда в скрипте много сложных вычислений или ошибка, ведущая к бесконечному циклу.

**Разница между микро и макрозадачами в event loop?**

В большинстве браузеров есть цикл событий для каждой вкладки браузера, чтобы изолировать каждый процесс и избежать веб-страниц с бесконечными циклами или тяжелой обработкой, что могут заблокировать весь твой браузер.

Идея Event loop очень проста. Есть бесконечный цикл, в котором движок JavaScript ожидает задачи, исполняет их и снова ожидает появления новых.

Задачи в очереди не обрабатываются одинаково. Они делятся на микрозадачи и макрозадания. Вместо того чтобы смешиваться, эти два типа задач занимают разные очереди.

Возникает вопрос, как цикл событий будет определять, какая очередь будет исключена из очереди и помещена в стек вызовов, когда он пуст. Решение зависит от следующих правил.

Одна макро-задача выполняется из очереди макро-задачи (Task) внутри очереди. По завершении этой задачи цикл событий переходит в очередь микрозадач (Job). «Цикл событий» не просматривает следующее действие до завершения всей очереди микрозадач (заданий). Он завершает всю очередь микро-задач, а затем возвращается в очередь макро-задач.

Очередь задач → Микро-задача → Рендеринг → Макро-задача

Это сделано так потому, что действия пользователя более важны, чем любая фоновая задача, такая как setTimeout, для обеспечения наилучшего взаимодействия с пользователем.

Более упрощённый алгоритм событийного цикла:

* выбрать и исполнить самую старую задачу из очереди макрозадач (например, «script»);
* выполнить все микрозадачи;
* пока очередь микрозадач не пуста; выбрать из очереди и исполнить самую старую микрозадачу;
* отрисовать изменения страницы, если они есть;
* если очередь макрозадач пуста – подождать, пока появится макрозадача;
* Перейти к шагу 1.

Любой код JavaScript, которому требуется слишком много времени для возврата контроля циклу событий, будет блокировать выполнение любого кода JavaScript на странице, даже блокировать поток пользовательского интерфейса, и пользователь не сможет кликать мышью, прокручивать страницу и т. д.

Подробнее о Event loop [**здесь**](https://learn.javascript.ru/event-loop).

Разница между ними в том, что у микрозадач есть приоритет перед макрозадачами. Они выполняются только после того, как выполнились все микрозадачи. Event loop переходит в очередь макрозадач — и затем снова после макрозадачи выполняются все микро-.

Помимо *макрозадач*, описанных в этой части, существуют *микрозадачи*, упомянутые в главе [Микрозадачи](https://learn.javascript.ru/microtask-queue).

Микрозадачи приходят только из кода. Обычно они создаются промисами: выполнение обработчика .then/catch/finally становится микрозадачей. Микрозадачи также используются «под капотом» await, т.к. это форма обработки промиса.

Также есть специальная функция queueMicrotask(func), которая помещает func в очередь микрозадач.

**Сразу после каждой *макрозадачи* движок исполняет все задачи из очереди *микрозадач* перед тем, как выполнить следующую макрозадачу или отобразить изменения на странице, или сделать что-то ещё.**

Например:

setTimeout(() => alert("timeout"));

Promise.resolve()

.then(() => alert("promise"));

alert("code");

Какой здесь будет порядок?

1. code появляется первым, т.к. это обычный синхронный вызов.
2. promise появляется вторым, потому что .then проходит через очередь микрозадач и выполняется после текущего синхронного кода.
3. timeout появляется последним, потому что это макрозадача.

Более подробное изображение событийного цикла выглядит так:

**Все микрозадачи завершаются до обработки каких-либо событий или рендеринга, или перехода к другой макрозадаче.**

Это важно, так как гарантирует, что общее окружение остаётся одним и тем же между микрозадачами – не изменены координаты мыши, не получены новые данные по сети и т.п.

Если мы хотим запустить функцию асинхронно (после текущего кода), но до отображения изменений и до новых событий, то можем запланировать это через queueMicrotask.

**Расскажите о queueMicrotask?**

Также есть специальная функция queueMicrotask(func), которая помещает func в очередь микрозадач.

<https://habr.com/ru/articles/681882/>

The **queueMicrotask()** method, which is exposed on the [Window](https://developer.mozilla.org/en-US/docs/Web/API/Window) or [Worker](https://developer.mozilla.org/en-US/docs/Web/API/Worker) interface, queues a microtask to be executed at a safe time prior to control returning to the browser's event loop.

The microtask is a short function which will run after the current task has completed its work and when there is no other code waiting to be run before control of the execution context is returned to the browser's event loop.

This lets your code run without interfering with any other, potentially higher priority, code that is pending, but before the browser regains control over the execution context, potentially depending on work you need to complete. You can learn more about how to use microtasks and why you might choose to do so in our [microtask guide](https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API/Microtask_guide).

The importance of microtasks comes in its ability to perform tasks asynchronously but in a specific order. See [Using microtasks in JavaScript with queueMicrotask()](https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API/Microtask_guide) for more details.

Microtasks are especially useful for libraries and frameworks that need to perform final cleanup or other just-before-rendering tasks.

**[Syntax](https://developer.mozilla.org/en-US/docs/Web/API/queueMicrotask" \l "syntax)**

queueMicrotask(() => {/\* ... \*/})

Copy to Clipboard

**[Parameters](https://developer.mozilla.org/en-US/docs/Web/API/queueMicrotask" \l "parameters)**

function

A [function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function) to be executed when the browser engine determines it is safe to call your code. Enqueued microtasks are executed after all pending tasks have completed but before yielding control to the browser's event loop.

**[Return value](https://developer.mozilla.org/en-US/docs/Web/API/queueMicrotask" \l "return_value)**

None ([undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined)).

[ДОКА](https://doka.guide/js/queuemicrotask/" \l ":~:text=queueMicrotask() добавляет переданную функцию в,запускается следующая функция в очереди.)

**Что такое промисы (Promises)?**

<https://www.w3schools.com/js/js_promise.asp>

<https://metanit.com/web/javascript/17.1.php>

<https://learn.javascript.ru/promise-basics>

Интерфейс **Promise** (промис) представляет собой обёртку для значения, неизвестного на момент создания промиса. Он позволяет обрабатывать результаты асинхронных операций так, как если бы они были синхронными: вместо конечного результата асинхронного метода возвращается своего рода *обещание* (дословный перевод слова "промис") получить результат в некоторый момент в будущем.

Promise может находиться в трёх состояниях:

* *ожидание (pending)*: начальное состояние, не исполнен и не отклонён.
* *исполнено (fulfilled)*: операция завершена успешно.
* *отклонено (rejected)*: операция завершена с ошибкой.

При создании промис находится в *ожидании (pending)*, а затем может стать *исполненным* (*fulfilled)*, вернув полученный результат (значение), или *отклонённым* (*rejected),* вернув причину отказа. В любом из этих случаев вызывается обработчик, прикреплённый к промису методом then. (Если в момент назначения обработчика промис уже исполнен или отклонён, обработчик всё равно будет вызван, т.е. асинхронное исполнение промиса и назначение обработчика не будет происходить в «состоянии гонки», как, например, в случае с событиями в DOM.)

Так как методы [Promise.prototype.then()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) и [Promise.prototype.catch()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch) сами возвращают промис, их можно вызывать цепочкой, создавая *соединения.*



**Примечание:** говорят, что промис находится в состоянии *завершён (settled)* когда он или исполнен или отклонён, т.е. в любом состоянии, кроме ожидания (это лишь форма речи, не являющаяся настоящим состоянием промиса). Также можно встретить термин *исполнен (resolved)* — это значит что промис *завершён* или "заблокирован" в ожидании завершения другого промиса. В статье [состояния и fates](https://github.com/domenic/promises-unwrapping/blob/master/docs/states-and-fates.md) приводится более подробное описание терминологии.

**[Свойства](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise" \l "свойства)**

Promise.length

Значение свойства всегда равно 1 (количество аргументов конструктора).

[Promise.prototype (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise)

Представляет прототип для конструктора Promise.

**[Методы](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise" \l "методы)**

[Promise.all(iterable)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise/all)

Ожидает исполнения всех промисов или отклонения любого из них.

Возвращает промис, который исполнится после исполнения всех промисов в iterable. В случае, если любой из промисов будет отклонён, Promise.all будет также отклонён.

[Promise.allSettled(iterable)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled)

Ожидает завершения всех полученных промисов (как исполнения так и отклонения).

Возвращает промис, который исполняется когда все полученные промисы завершены (исполнены или отклонены), содержащий массив результатов исполнения полученных промисов.

[Promise.race(iterable)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise/race)

Ожидает исполнения или отклонения любого из полученных промисов.

Возвращает промис, который будет исполнен или отклонён с результатом исполнения первого исполненного или отклонённого промиса из iterable.

[Promise.reject(reason)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise/reject)

Возвращает промис, отклонённый из-за reason.

[Promise.resolve(value)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve)

Возвращает промис, исполненный с результатом value.

**[Создание промиса](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise" \l "создание_промиса)**

Объект Promise создаётся при помощи ключевого слова new и своего конструктора. Конструктор Promise принимает в качестве аргумента функцию, называемую "исполнитель" (*executor function*). Эта функция должна принимать две функции-колбэка в качестве параметров. Первый из них (resolve) вызывается, когда асинхронная операция завершилась успешно и вернула результат своего исполнения в виде значения. Второй колбэк (reject) вызывается, когда операция не удалась, и возвращает значение, указывающее на причину неудачи, чаще всего объект ошибки.

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

// выполняется асинхронная операция, которая в итоге вызовет:

//

// resolve(someValue); // успешное завершение

// или

// reject("failure reason"); // неудача

});

Чтобы снабдить функцию функциональностью промисов, нужно просто вернуть в ней объект Promise:

function myAsyncFunction(url) {

return new Promise((resolve, reject) => {

const xhr = new XMLHttpRequest();

xhr.open("GET", url);

xhr.onload = () => resolve(xhr.responseText);

xhr.onerror = () => reject(xhr.statusText);

xhr.send();

});

}

**[Примеры](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise" \l "примеры)**

**[Простой пример](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise" \l "простой_пример)**

Создать промис!

let myFirstPromise = new Promise((resolve, reject) => {

// Мы вызываем resolve(...), когда асинхронная операция завершилась успешно, и reject(...), когда она не удалась.

// В этом примере мы используем setTimeout(...), чтобы симулировать асинхронный код.

// В реальности вы, скорее всего, будете использовать XHR, HTML5 API или что-то подобное.

setTimeout(function(){

resolve("Success!"); // Ура! Всё прошло хорошо!

}, 250);

});

myFirstPromise.then((successMessage) => {

// successMessage - это что угодно, что мы передали в функцию resolve(...) выше.

// Это необязательно строка, но если это всего лишь сообщение об успешном завершении, это наверняка будет она.

console.log("Ура! " + successMessage);

});

Данный небольшой пример показывает механизм работы с Promise. Метод testPromise() вызывается при каждом нажатии на [<button>](https://developer.mozilla.org/ru/docs/Web/HTML/Element/button). При этом создаётся промис, который успешно выполняется при помощи window.setTimeout, со значением 'result' в случайном интервале от 1 до 3-х секунд.

исполнение промиса протоколируется при помощи продолжения p1.then. Это показывает как синхронная часть метода отвязана от асинхронного завершения промиса.

[DOKA](https://doka.guide/js/promise/)

**Плюсы и минусы использовании Ajax?**

**Преимущества**

Для продвижения сайта применение AJAX имеет ряд преимуществ:

* экономия [трафика](https://ingate.ru/uslugi/seo-prodvizhenie-saita-po-trafiku/) пользователя (вместо обновления всей страницы, загружается ее небольшая изменившаяся часть);
* снижение нагрузки на сервер. К примеру, на странице личных сообщений форума при выделении пользователем прочитанных писем сервер вносит изменения в БД и отправляет скрипту клиента ответ о выполнении операции без повторного создания страницы и ее передачи;
* ускорение реагирования интерфейса на команды пользователя.

**Недостатки**

При поисковой [оптимизации сайта](https://ingate.ru/uslugi/seo/) учитывают следующие недостатки AJAX:

* не всегда возможна интеграция со стандартным набором инструментов браузера. Так как интернет-обозреватели не регистрируют в истории переходы по страницам, нельзя воспользоваться кнопкой «Назад». В некоторых случаях нет возможности добавить в закладки нужный материал;
* контент, загружаемый динамически, не доступен поисковым системам, поэтому необходимо обеспечить альтернативный доступ к содержимому ресурса;
* неправильный учет статистики перемещения пользователя по сайту;
* усложнение контроля целостности типов и форматов, так как процессы форматирования данных частично переносятся на сторону клиента;
* в браузере пользователя должен быть включен JavaScript.

Альтернативой AJAX выступают Java-аплеты, JavaFX, технологии ActionScript 3, Flash Remoting, Adobe Flex, составляющие технологическую основу Rich Internet Applications от Macromedia, и Silverlight от корпорации Microsoft.

**Подходы при работе с асинхронным кодом?**

Есть два типа стиля асинхронного кода, с которыми вы столкнётесь в коде JavaScript, старый метод — колбэки (callbacks) и более новый — промисы (promises).

Асинхронные колбэки — это функции, которые определяются как аргументы при вызове функции, которая начнёт выполнение кода на заднем фоне. Когда код на заднем фоне завершает свою работу, он вызывает колбэк-функцию, оповещающую, что работа сделана, либо оповещающую о трудностях в завершении работы. Обратные вызовы — немного устаревшая практика, но они все ещё употребляются в некоторых старомодных, но часто используемых API.

Пример асинхронного колбэка вторым параметром [addEventListener()](https://developer.mozilla.org/ru/docs/Web/API/EventTarget/addEventListener) (как мы видели выше):

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

alert('You clicked me!');

let pElem = document.createElement('p');

pElem.textContent = 'This is a newly-added paragraph.';

document.body.appendChild(pElem);

});

Первый параметр — тип обрабатываемого события, второй параметр — колбэк-функция, вызываемая при срабатывании события.

При передаче колбэк-функции как аргумента в другую функцию, мы передаём только ссылку на функцию как аргумент, следовательно колбэк-функция **не** выполняется мгновенно. Она вызывается асинхронно внутри тела, содержащего функцию. Эта функция должна выполнять колбэк-функцию в нужный момент.

Колбэк-функции универсальны — они не только позволяют вам контролировать порядок, в котором запускаются функции и данные, передающиеся между ними, они также позволяют передавать данные различным функциям, в зависимости от обстоятельств. Вы можете выполнять различные действия с загруженным ответом, такие как processJSON(), displayText(), и другие.

Заметьте, что не все колбэк-функции асинхронны — некоторые запускаются синхронно. Например, при использовании [Array.prototype.forEach()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach) для перебора элементов массива ([запустите пример](https://mdn.github.io/learning-area/javascript/asynchronous/introducing/foreach.html), и [посмотрите исходный код](https://github.com/mdn/learning-area/blob/master/javascript/asynchronous/introducing/foreach.html)):

Промисы — новый стиль написания асинхронного кода, который используется в современных Web API. Хорошим примером является [fetch()](https://developer.mozilla.org/en-US/docs/Web/API/fetch) API, который современнее и эффективнее чем [XMLHttpRequest](https://developer.mozilla.org/ru/docs/Web/API/XMLHttpRequest). Посмотрим на краткий пример, из нашей статьи [Fetching data from the server](https://developer.mozilla.org/ru/docs/Learn/JavaScript/Client-side_web_APIs/Fetching_data):

fetch('products.json')

.then( response => {

if (!response.ok) {

throw new Error(`HTTP error: ${response.status}`);

}

return response.json();

})

.then( json => initialize(json) )

.catch( err => console.error(`Fetch problem: ${err.message}`) );

В примере видно, как fetch() принимает один параметр — URL ресурса, который нужно получить из сети, — и возвращает [промис](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise). Промис — это объект, представляющий асинхронную операцию, выполненную удачно или неудачно. Он представляет собой как бы промежуточное состояние. По сути, это способ браузера сказать: "я обещаю вернуться к вам с ответом как можно скорее", поэтому в дословном переводе "промис" (promise) означает "обещание".

Может понадобиться много времени, чтобы привыкнуть к данной концепции; это немного напоминает [Кота Шрёдингера](https://ru.wikipedia.org/wiki/Кот_Шрёдингера) в действии. Ни один из возможных результатов ещё не произошёл, поэтому операция fetch в настоящее время ожидает результата. Далее у нас есть три блока кода следующих сразу после fetch():

* Два [then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) блока. Оба включают в себя функцию обратного вызова, которая запустится, если предыдущая операция закончилась успешно, и каждая колбэк-функция принимает на вход результат предыдущей успешно выполненной операции, таким образом вы можете выполнять операции последовательно. Каждый .then() блок возвращает новый promise, это значит что вы можете объединять в цепочки блоки .then(), таким образом можно выполнить несколько асинхронных операций по порядку, одну за другой.
* [catch()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch) блок описывается в конце и будет запущен если какой-либо .then() блок завершится с ошибкой — это аналогично синхронному [try...catch](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch), ошибка становится доступной внутри catch(), что может быть использовано для сообщения пользователю о типе возникшей ошибки. Однако синхронный try...catch не будет работать с promise, хотя будет работать с [async/await](https://developer.mozilla.org/ru/docs/Learn/JavaScript/Asynchronous/Promises), с которыми вы познакомитесь позже.

Промисы имеют некоторое сходство со старомодными колбэк-функциями. По сути, они являются возвращаемым объектом, к которому вы присоединяете колбэк-функции, вместо того, чтобы передавать колбэки в функцию.

Тем не менее, промисы сделаны специально для обработки асинхронных операций, и имеют много преимуществ по сравнению с колбэками:

* Вы можете объединить несколько асинхронных операций вместе, используя несколько операций.then(), передавая результат одного в следующий в качестве входных данных. Это гораздо сложнее сделать с колбэками, которые часто заканчиваются массивным «адом колбэков» (также известным как [callback hell](http://callbackhell.com/)).
* Обратные вызовы Promise всегда вызываются в строгом порядке, который они помещают в очередь событий..
* Обработка ошибок намного лучше — все ошибки обрабатываются одним блоком .catch () в конце блока, а не обрабатываются индивидуально на каждом уровне «пирамиды».
* Промисы избегают инверсии управления, в отличие от колбэков, которые теряют полный контроль над тем, как будет выполняться функция при передаче колбэка в стороннюю библиотеку.

[More info](https://developer.mozilla.org/ru/docs/Learn/JavaScript/Asynchronous/Introducing)

[Habr](https://habr.com/ru/companies/ruvds/articles/337662/)

<https://ivan-shamaev.ru/asynchronous-javascript-callbacks-promises-and-async-await/>

**Преимущества использовании промисов вместо колбэков?**

Какие преимущества и недостатки при использовании промисов вместо колбэков (callbacks)?

Преимущества

* Помогает избежать “callback hell”, который может быть нечитаемым.
* Упрощает написание последовательного удобочитаемого асинхронного кода с помощью then().
* Упрощает написание параллельного асинхронного кода с помощью Promise.all().
* С использованием промисов можно избежать следующих проблем: колбэк-функция была вызвана слишком рано, поздно или вовсе не была вызвана; функция была вызвана слишком мало или слишком много раз; не удалось передать необходимую среду/параметры; были пропущены ошибки/исключения.

Недостатки

* Чуть более сложный код.
* В старых браузерах, где не поддерживается ES2015, нужно загрузить полифил, чтобы их использовать.

**Что такое callback-функция? Что такое Callback Hell?**

**Что такое коллбэк?**

**Простыми словами:** коллбэк — это функция, которая должна быть выполнена после того, как другая функция завершила выполнение (отсюда и название: callback — функция обратного вызова).

**Чуть сложнее:** В JavaScript функции — это объекты. Поэтому функции могут принимать другие функции в качестве аргументов, а также функции могут возвращать функции в качестве результата. Функции, которые это умеют, называются **функциями высшего порядка**. А любая функция, которая передается как аргумент, называется **callback-функцией**. Чтобы лучше разобраться, давайте посмотрим на примерах, как это выглядит.

**Зачем нам нужны коллбэки?**

По одной простой причине — JavaScript это событийно-ориентированный язык. Это значит, что вместо того, чтобы ждать ответа для дальнейшего выполнения программы, JavaScript продолжит выполнение, одновременно ожидая других событий. Давайте разберем простой пример:

function first(){  
 console.log(1);  
}  
function second(){  
 console.log(2);  
}  
first();  
second();

Как вы и ожидаете, функция first выполнится первой, а функция second после нее, и в консоли будет выведен следующий результат:

// 1  
// 2

Пока что все понятно. Но что если функция first содержит некий код, который не может выполнится немедленно? К примеру, запрос к API, где мы отправляем запрос и должны ждать ответа. Чтобы смоделировать такую ситуацию, мы используем функцию setTimeout, которая вызывает функцию после заданного временного промежутка. Мы отсрочим выполнение функции на 500 миллисекунд, как будто бы это запрос к некому API. Теперь код будет выглядеть так:

function first(){  
 // Как будто бы запрос к API  
 setTimeout( function(){  
 console.log(1);  
 }, 500 );  
}  
function second(){  
 console.log(2);  
}  
first();  
second();

Неважно, понимаете ли вы сейчас, как работает setTimout(). Основная идея – теперь мы отложили исполнение команды console.log(1) на 500 миллисекунд. И что теперь выведет наша программа?

first();  
second();  
// 2  
// 1

Хотя мы по-прежнему вызываем функцию first первой, ее вывод появился вторым, после вывода функции second. Но JavaScript не нарушает порядок вызова функций, он просто не дожидается ответа от функции first, а сразу двигается дальше – к функции second.

Зачем я вам это показал? Чтобы вы понимали, нельзя просто вызывать функции в нужном порядке и надеяться, что они в любом случае выполнятся в том же порядке. Коллбэки же позволяют нам быть уверенными в том, что определенный код не начнет исполнение до того момента, пока другой код не завершит исполнение.

**Создаем callback**

Хватит болтовни, теперь давайте создадим коллбэк.

Во-первых, откройте консоль разработчика в Google Chrome (Windows: Ctrl + Shift + J)(Mac: Cmd + Option + J) и введите в консоли следующую функцию:

function doHomework(subject) {  
 alert(`Starting my ${subject} homework.`);  
}

Мы создали функцию doHomework. Наша функция принимает одну переменную – название предмета, которым мы будем заниматься. Вызовите функцию, набрав следующий текст в консоли:

doHomework('math');  
// Выводит алерт: Starting my math homework.

Теперь давайте добавим в определение функции еще один параметр, это и будет наш коллбэк. Затем вызовем ее, определив функцию-callback в качестве аргумента:

function doHomework(subject, callback) {  
 alert(`Starting my ${subject} homework.`);  
 callback();  
}doHomework('math', function() {  
 alert('Finished my homework');  
});

Если вы введете этот код в консоли, вы получите два алерта один за другим, в первом будет сообщение о том, что выполнение домашнего задания началось (Starting my math homework.), а во втором — что вы закончили выполнять задание (Finished my homework).

Однако коллбэки не обязательно быть определены при вызове функции. Они могут быть определены и в другом месте кода, например, так:

function doHomework(subject, callback) {  
 alert(`Starting my ${subject} homework.`);  
 callback();  
}  
function alertFinished(){  
 alert('Finished my homework');  
}  
doHomework('math', alertFinished);

Таким образом, результат выполнения этого кода, такой же, как и в предыдущем примере, однако сам код немного другой. Как вы видите, мы передали функцию alertFinished как аргумент в функцию doHomework при ее вызове.

**Пример из реальной жизни**

На прошлой неделе я опубликовал статью [«Создаем бота для Твиттера в 38 строк кода»](https://codeburst.io/build-a-simple-twitter-bot-with-node-js-in-just-38-lines-of-code-ed92db9eb078). Этот код работает благодаря API Твиттера. И когда мы делаем запрос к API, мы должны дождаться ответа до того, как начнем выполнять с этим ответом какие-то действия. Это прекрасный пример того, как в реальной жизни выглядит callback-функция. Вот как выглядит сам запрос:

T.get('search/tweets', params, function(err, data, response) {  
 if(!err){  
 // Происходит какая-то магия  
 } else {  
 console.log(err);  
 }  
})

T.get просто значит, что мы выполняем get запрос к API Твиттера. В запросе три параметра: 'search/tweets' – это адрес (роут) запроса, params – наши параметры поиска и в конце передается анонимная функция-callback.

Коллбэк здесь нужен, потому что нам нужно дождаться ответа от сервера до того, как приступим к дальнейшему выполнению кода. Мы не знаем, успешным будет наш запрос или нет, поэтому после отправки параметров поиска на search/tweets через get-запрос, мы просто ждем. Как только Твиттер ответит, выполнится наша callback-функция. Твиттер отправит нам в качестве ответа или объект err (error – ошибка), или объект response. В коллбэке мы можем через if() проверить, был ли запрос успешным или нет, и затем действовать соответственно.

[About callback hell](http://callbackhell.ru/" \l ":~:text=Это ласково называют адом обратных,Многие разработчики делают эту ошибку.)

<https://howtojs.ru/callback-hell/>

**Что такое async/await?**

Существует специальный синтаксис для работы с промисами, который называется «async/await». Он удивительно прост для понимания и использования.

**[Асинхронные функции](https://learn.javascript.ru/async-await" \l "asinhronnye-funktsii)**

Начнём с ключевого слова async. Оно ставится перед функцией, вот так:

async function f() {

return 1;

}

У слова async один простой смысл: эта функция всегда возвращает промис. Значения других типов оборачиваются в завершившийся успешно промис автоматически.

Например, эта функция возвратит выполненный промис с результатом 1:

async function f() {

return 1;

}

f().then(alert); // 1

Можно и явно вернуть промис, результат будет одинаковым:

async function f() {

return Promise.resolve(1);

}

f().then(alert); // 1

Так что ключевое слово async перед функцией гарантирует, что эта функция в любом случае вернёт промис. Согласитесь, достаточно просто? Но это ещё не всё. Есть другое ключевое слово – await, которое можно использовать только внутри async-функций.

**[Await](https://learn.javascript.ru/async-await" \l "await)**

Синтаксис:

// работает только внутри async–функций

let value = await promise;

Ключевое слово await заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от await не выполнится. После чего оно вернёт его результат, и выполнение кода продолжится.

В этом примере промис успешно выполнится через 1 секунду:

async function f() {

let promise = new Promise((resolve, reject) => {

setTimeout(() => resolve("готово!"), 1000)

});

let result = await promise; // будет ждать, пока промис не выполнится (\*)

alert(result); // "готово!"

}

f();

В данном примере выполнение функции остановится на строке (\*) до тех пор, пока промис не выполнится. Это произойдёт через секунду после запуска функции. После чего в переменную result будет записан результат выполнения промиса, и браузер отобразит alert-окно «готово!».

Обратите внимание, хотя await и заставляет JavaScript дожидаться выполнения промиса, это не отнимает ресурсов процессора. Пока промис не выполнится, JS-движок может заниматься другими задачами: выполнять прочие скрипты, обрабатывать события и т.п.

По сути, это просто «синтаксический сахар» для получения результата промиса, более наглядный, чем promise.then.

**await нельзя использовать в обычных функциях**

Если мы попробуем использовать await внутри функции, объявленной без async, получим синтаксическую ошибку:

function f() {

let promise = Promise.resolve(1);

let result = await promise; // SyntaxError

}

Ошибки не будет, если мы укажем ключевое слово async перед объявлением функции. Как было сказано раньше, await можно использовать только внутри async–функций.

Давайте перепишем пример showAvatar() из раздела [Цепочка промисов](https://learn.javascript.ru/promise-chaining) с помощью async/await:

1. Нам нужно заменить вызовы .then на await.
2. И добавить ключевое слово async перед объявлением функции.

async function showAvatar() {

// запрашиваем JSON с данными пользователя

let response = await fetch('/article/promise-chaining/user.json');

let user = await response.json();

// запрашиваем информацию об этом пользователе из github

let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);

let githubUser = await githubResponse.json();

// отображаем аватар пользователя

let img = document.createElement('img');

img.src = githubUser.avatar\_url;

img.className = "promise-avatar-example";

document.body.append(img);

// ждём 3 секунды и затем скрываем аватар

await new Promise((resolve, reject) => setTimeout(resolve, 3000));

img.remove();

return githubUser;

}

showAvatar();

Получилось очень просто и читаемо, правда? Гораздо лучше, чем раньше.

**await нельзя использовать на верхнем уровне вложенности**

Программисты, узнав об await, часто пытаются использовать эту возможность на верхнем уровне вложенности (вне тела функции). Но из-за того, что await работает только внутри async–функций, так сделать не получится:

// SyntaxError на верхнем уровне вложенности

let response = await fetch('/article/promise-chaining/user.json');

let user = await response.json();

Можно обернуть этот код в анонимную async–функцию, тогда всё заработает:

(async () => {

let response = await fetch('/article/promise-chaining/user.json');

let user = await response.json();

...

})();

**await работает с «thenable»–объектами**

Как и promise.then, await позволяет работать с промис–совместимыми объектами. Идея в том, что если у объекта можно вызвать метод then, этого достаточно, чтобы использовать его с await.

В примере ниже, экземпляры класса Thenable будут работать вместе с await:

class Thenable {

constructor(num) {

this.num = num;

}

then(resolve, reject) {

alert(resolve);

// выполнить resolve со значением this.num \* 2 через 1000мс

setTimeout(() => resolve(this.num \* 2), 1000); // (\*)

}

};

async function f() {

// код будет ждать 1 секунду,

// после чего значение result станет равным 2

let result = await new Thenable(1);

alert(result);

}

f();

Когда await получает объект с .then, не являющийся промисом, JavaScript автоматически запускает этот метод, передавая ему аргументы – встроенные функции resolve и reject. Затем await приостановит дальнейшее выполнение кода, пока любая из этих функций не будет вызвана (в примере это строка (\*)). После чего выполнение кода продолжится с результатом resolve или reject соответственно.

**Асинхронные методы классов**

Для объявления асинхронного метода достаточно написать async перед именем:

class Waiter {

async wait() {

return await Promise.resolve(1);

}

}

new Waiter()

.wait()

.then(alert); // 1

Как и в случае с асинхронными функциями, такой метод гарантированно возвращает промис, и в его теле можно использовать await.

**[Обработка ошибок](https://learn.javascript.ru/async-await" \l "obrabotka-oshibok)**

Когда промис завершается успешно, await promise возвращает результат. Когда завершается с ошибкой – будет выброшено исключение. Как если бы на этом месте находилось выражение throw.

Такой код:

async function f() {

await Promise.reject(new Error("Упс!"));

}

Делает то же самое, что и такой:

async function f() {

throw new Error("Упс!");

}

Но есть отличие: на практике промис может завершиться с ошибкой не сразу, а через некоторое время. В этом случае будет задержка, а затем await выбросит исключение.

Такие ошибки можно ловить, используя try..catch, как с обычным throw:

async function f() {

try {

let response = await fetch('http://no-such-url');

} catch(err) {

alert(err); // TypeError: failed to fetch

}

}

f();

В случае ошибки выполнение try прерывается и управление прыгает в начало блока catch. Блоком try можно обернуть несколько строк:

async function f() {

try {

let response = await fetch('/no-user-here');

let user = await response.json();

} catch(err) {

// перехватит любую ошибку в блоке try: и в fetch, и в response.json

alert(err);

}

}

f();

Если у нас нет try..catch, асинхронная функция будет возвращать завершившийся с ошибкой промис (в состоянии rejected). В этом случае мы можем использовать метод .catch промиса, чтобы обработать ошибку:

async function f() {

let response = await fetch('http://no-such-url');

}

// f() вернёт промис в состоянии rejected

f().catch(alert); // TypeError: failed to fetch // (\*)

Если забыть добавить .catch, то будет сгенерирована ошибка «Uncaught promise error» и информация об этом будет выведена в консоль. Такие ошибки можно поймать глобальным обработчиком, о чём подробно написано в разделе [Промисы: обработка ошибок](https://learn.javascript.ru/promise-error-handling).

**async/await и promise.then/catch**

При работе с async/await, .then используется нечасто, так как await автоматически ожидает завершения выполнения промиса. В этом случае обычно (но не всегда) гораздо удобнее перехватывать ошибки, используя try..catch, нежели чем .catch.

Но на верхнем уровне вложенности (вне async–функций) await использовать нельзя, поэтому .then/catch для обработки финального результата или ошибок – обычная практика.

Так сделано в строке (\*) в примере выше.

**async/await отлично работает с Promise.all**

Когда необходимо подождать несколько промисов одновременно, можно обернуть их в Promise.all, и затем await:

// await будет ждать массив с результатами выполнения всех промисов

let results = await Promise.all([

fetch(url1),

fetch(url2),

...

]);

В случае ошибки она будет передаваться как обычно: от завершившегося с ошибкой промиса к Promise.all. А после будет сгенерировано исключение, которое можно отловить, обернув выражение в try..catch.

**[Итого](https://learn.javascript.ru/async-await" \l "itogo)**

Ключевое слово async перед объявлением функции:

1. Обязывает её всегда возвращать промис.
2. Позволяет использовать await в теле этой функции.

Ключевое слово await перед промисом заставит JavaScript дождаться его выполнения, после чего:

1. Если промис завершается с ошибкой, будет сгенерировано исключение, как если бы на этом месте находилось throw.
2. Иначе вернётся результат промиса.

Вместе они предоставляют отличный каркас для написания асинхронного кода. Такой код легко и писать, и читать.

Хотя при работе с async/await можно обходиться без promise.then/catch, иногда всё-таки приходится использовать эти методы (на верхнем уровне вложенности, например). Также await отлично работает в сочетании с Promise.all, если необходимо выполнить несколько задач параллельно.

<https://tproger.ru/articles/uglubljaemsja-v-javascript-vsjo-li-mozhet-async-await-ili-kogda-ispolzovat-promise/>

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

В чем разница между Promise и async Await?

Async создает Promise. А await ждет выполнения промиса. Promise — обертка (класс, для простоты понимания) для отложенных и асинхронных вычислений. Ожидает выполнения колбэк функций и никак иначе.

**Разница между Promise.all(), Promise.any() и Promise.race()?**

Функции Promise.all(), Promise.allSettled() и Promise.race() позволяют сгруппировать выполнение нескольких промисов.

Функция Promise.all

Функция Promise.all() возвращает единый объект Promise, который объединяет набор промисов.

Рассмотрим следуюший код:

const promise1 = new Promise((resolve,reject) => {

setTimeout(resolve, 1000, "Hello");

});

const promise2 = new Promise((resolve,reject) => {

setTimeout(resolve, 500, "World");

});

promise1.then(value => console.log(value)); // Hello

promise2.then(value => console.log(value)); // World

Здесь определено два промиса. Асинхронная операция первого промиса выполняется через 1000 миллисекунд, действие второго промиса выполняется через 500 миллисекунд. Оба этих промиса выполняются независимо друг от друга. Консольный вывод:

World

Hello

Функция Promise.all() позволяет объединить несколько промисов и выполнять их параллельно, но как единое целое. В качестве параметра функция принимает набор промисов:

Promise.all([промис1, промис2, ... прромисN]);

Возвращаемым результатом функции является новый объект Promise.

Теперь изменим предыдущий пример, использовав функцию Promise.all():

const promise1 = new Promise((resolve,reject) => {

setTimeout(resolve, 1000, "Hello");

});

const promise2 = new Promise((resolve,reject) => {

setTimeout(resolve, 500, "World");

});

Promise.all([promise1, promise2])

.then(values => {

const [promise1data, promise2data] = values;

console.log(promise1data, promise2data); // Hello World

});

Теперь данные обоих промисов возвращаются вместе и доступны в методе then() в виде массива values. Консольный вывод:

Hello World

Значения всех промисов возвращаются только если все они завершились успешно. Но если в асинхронной операции хотя бы одного промиса возникнет ошибка в силу внутренней логики или из-за вызова функции reject(), то все промисы перейдут в состояние rejected, соответственно:

const promise1 = new Promise((resolve,reject) => {

reject("Непредвиденная ошибка");

setTimeout(resolve, 500, "Hello");

});

const promise2 = new Promise((resolve,reject) => {

setTimeout(resolve, 1000, "World");

});

Promise.all([promise1, promise2])

.then(values => {

const [promise1data, promise2data] = values;

console.log(promise1data, promise2data);

})

.catch(error => console.log(error)); // Непредвиденная ошибка

В данном случае мы явным образом переводим переводим первый промис в состояние rejected, соответственно в это состояние переводятся все промисы, переданные в функцию Promise.all(). И далее через метод catch(), как и в обзем случае, мы можем обработать возникшую ошибку.

Promise.allSettled

Еще одна функция - Promise.allSettled() также как и Promise.all() принимает набор промисов и выполняет их как единое целое, но возвращает объект со статусом и результатом промиса:

const promise1 = new Promise((resolve,reject) => {

reject("Непредвиденная ошибка");

setTimeout(resolve, 500, "Hello");

});

const promise2 = new Promise((resolve,reject) => {

setTimeout(resolve, 1000, "World");

});

Promise.allSettled([promise1, promise2])

.then(values => {

const [promise1data, promise2data] = values;

console.log(promise1data); // {status: "rejected", reason: "Непредвиденная ошибка"}

console.log(promise2data); // {status: "fulfilled", value: "World"}

});

При этом при возникновении ошибки в одном из промисов (как в случае выше с первым промисом) функция также передается результаты в метод then(), который следует дальше в цепочке. Каждый результат представляет объект. Первое свойство этого объекта - status описывает статус или состояние промиса. Если произошла ошибка, статус rejected, а второе свойство представляет объект ошибки. Если промис успешно завершил выполнение, то статус fulfilled, а второе свойство - value хранит результат промиса.

Promise.race

Функция Promise.race() также принимает несколько промисов, только возвращает первый завершенный промис (вне зависимости завершился от успешно или с ошибкой):

const promise1 = new Promise((resolve) => {

setTimeout(resolve, 500, "Hello");

});

const promise2 = new Promise((resolve) => {

setTimeout(resolve, 1000, "World");

});

Promise.race([promise1, promise2])

.then(value => console.log(value)) // Hello

.catch(error => console.log(error));

В данном случае первым выполненным будет промис promise1. Поэтому в метод then(value => console.log(value)) в качестве value будет передана строка "Hello".

Promise.any

Функция Promise.any() принимает несколько промисов и возвращает первый успешно завершившийся промис:

const promise1 = new Promise((resolve, reject) => {

reject("error in promise1");

setTimeout(resolve, 500, "Hello");

});

const promise2 = new Promise((resolve) => {

setTimeout(resolve, 1000, "World");

});

Promise.any([promise1, promise2])

.then(value => console.log(value)) // World

.catch(error => console.log(error));

В данном случае первым выполненным будет промис promise1, однако он завершается с ошибкой. Поэтому в метод then(value => console.log(value)) в качестве value будет передана строка "World" - результат промиса promise2, который успешно завершается.

Может показаться, что Promise.any() делает то же самое, что и Promise.race(), однако эти функции отличаются в отношении того, как они работают с промисами, которые завершились с ошибкой. Promise.race() возвращает первый завершенный промис, вне зависимости завершился он с ошибкой или нет. А Promise.any() возвращает первый успешно завершенный промис (если такой имеется). Если же все промисы завершились с ошибкой, то генерируется исключение типа AggregateError:

const promise1 = new Promise((resolve, reject) => {

reject("error in promise1");

setTimeout(resolve, 500, "Hello");

});

const promise2 = new Promise((resolve, reject) => {

reject("error in promise2");

setTimeout(resolve, 1000, "World");

});

Promise.any([promise1, promise2])

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

.catch(error => console.log(error)); // AggregateError: All promises were rejected

С помощью свойства errors типа AggregateError можно получить в виде массива все ошибки, которые возникли в промисах:

const promise1 = new Promise((resolve, reject) => {

reject("error in promise1");

setTimeout(resolve, 500, "Hello");

});

const promise2 = new Promise((resolve, reject) => {

reject("error in promise2");

setTimeout(resolve, 1000, "World");

});

Promise.any([promise1, promise2])

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

.catch(e => console.log(e.errors)); // ["error in promise1", "error in promise2"]

[Medium](https://medium.com/weekly-webtips/javascript-promises-promise-all-vs-promise-allsettled-vs-promise-race-vs-promise-any-d42b8e272bf8)

<https://learn.javascript.ru/promise-api>

**Расскажите про статический метод .allSettled()?**

Promise.all завершается с ошибкой, если она возникает в любом из переданных промисов. Это подходит для ситуаций «всё или ничего», когда нам нужны *все* результаты для продолжения:

Promise.all([

fetch('/template.html'),

fetch('/style.css'),

fetch('/data.json')

]).then(render); // методу render нужны результаты всех fetch

Метод Promise.allSettled всегда ждёт завершения всех промисов. В массиве результатов будет

* {status:"fulfilled", value:результат} для успешных завершений,
* {status:"rejected", reason:ошибка} для ошибок.

Например, мы хотели бы загрузить информацию о множестве пользователей. Даже если в каком-то запросе ошибка, нас всё равно интересуют остальные.

Используем для этого Promise.allSettled:

let urls = [

'https://api.github.com/users/iliakan',

'https://api.github.com/users/remy',

'https://no-such-url'

];

Promise.allSettled(urls.map(url => fetch(url)))

.then(results => { // (\*)

results.forEach((result, num) => {

if (result.status == "fulfilled") {

alert(`${urls[num]}: ${result.value.status}`);

}

if (result.status == "rejected") {

alert(`${urls[num]}: ${result.reason}`);

}

});

});

Массив results в строке (\*) будет таким:

[

{status: 'fulfilled', value: ...объект ответа...},

{status: 'fulfilled', value: ...объект ответа...},

{status: 'rejected', reason: ...объект ошибки...}

]

То есть, для каждого промиса у нас есть его статус и значение/ошибка.

[DOKA](https://doka.guide/js/promise-allsettled/" \l ":~:text=На практике,-Скопировать ссылку на&text=Он особенно удобен%2C когда запросы,all() .)

**Плюсы и минусы асинхронного программирования в JavaScript?**

**Плюсы асинхронного программирования**  
Существует распространенное заблуждение, что асинхронное программирование означает большую производительность и скорость. На самом деле выполнение нескольких задач одновременно не оказывает прямого влияния на эти области. Но у асинхронного программирования есть немало преимуществ, которые вы можете увидеть ниже.  
  
**Пользовательский опыт**При сравнении асинхронного и синхронного программирования асинхронные программы улучшают взаимодействие с пользователем различными способами. Во-первых, все сценарии программы или веб-страницы могут загружаться одновременно, что приводит к лучшему отклику и уменьшению задержек загрузки страницы. И сбой потока не влияет на возможность запуска или рендеринга других потоков. Следовательно, асинхронные приложения в целом более доступны.  
  
**Настройка**  
На практике асинхронное программирование требует написания функций обратного вызова в соответствии с ошибками, которые могут возникнуть и нарушить работу программы. Эти обратные вызовы предписывают машине пройти мимо ошибки и запустить следующую задачу. Это также дает программистам возможность написать персонализированное сообщение об ошибке. В результате ошибки не означают, что ваша программа вообще не запускается. Пользователи получают приятную заметку, в то время как все остальные функции программы работают нормально.  
  
**Масштабируемость**  
Async/await — это хорошо известная синтаксическая структура многих языков программирования, которая позволяет асинхронным функциям удобно работать с промисами. Промисы — это объекты, которые инкапсулируют желаемое поведение асинхронной операции. Обе эти концепции являются неотъемлемой частью понятия масштабируемости. Масштабируемость может осуществляться двумя способами — горизонтально и вертикально.  
  
Синхронные и асинхронные программы могут использовать горизонтальное масштабирование, добавляя дополнительные серверы для масштабирования служб. С другой стороны, вертикальное масштабирование — это более простой процесс, который могут использовать асинхронные программы, чтобы позволить одному серверу увеличить количество запросов, которые может обработать сервер. Подключая асинхронную функцию с помощью ключевого слова async/await, разработчики могут упростить программы, которые в противном случае были бы синхронными, используя методы обратного вызова на основе обещаний.  
  
**Минусы асинхронного программирования**  
Асинхронное программирование может показаться очевидным решением любых узких мест, которые могут возникнуть в ваших проектах разработки программного обеспечения. Но есть причины, по которым разработчики избегают использования асинхронного программирования.  
  
Сложность  
Чтобы успешно программировать асинхронные операции, разработчики должны иметь глубокие знания об обратных вызовах и рекурсивных функциях. И даже если они это сделают, программирование этого в приложении может быть громоздкой задачей во время разработки. Жизненный цикл разработки программного обеспечения в целом будет более сложным и заметно более медленным. По той же причине сам код может усложняться. Написание чистого кода становится более сложной задачей, а тестирование и отладка — такими же напряженными.  
  
Задержка  
Хотя обновление страницы является меньшей проблемой при асинхронном программировании по сравнению с синхронным программированием, первоначальный рендеринг страницы может занять некоторое время. Кроме того, слишком много асинхронных запросов могут перегрузить ваш сервер, и ваша программа может работать медленнее, несмотря на параллелизм, который вы получаете взамен .  
  
Совместимость  
C++ и JavaScript — самые выдающиеся языки программирования, поддерживающие асинхронное программирование. В этих языках ключевое слово async широко используется и почитается. Но с другими языками дело обстоит не так просто. Хотя, безусловно, можно программировать асинхронные программы практически на любом языке, это будет трудоемкой задачей, если такая реализация не будет предварительно оснащена рассматриваемым языком.  
  
**Что такое синхронное программирование?**  
Синхронные программы используют блокирующие операции ввода-вывода, в которых каждая операция должна выполняться до выполнения следующей. Как правило, компьютеры работают быстро, поэтому синхронная связь не обязательно невыгодна. Однако в реальном, нетехническом приложении это было бы эквивалентно отказу официанта принести какое-либо из заказанных вами блюд, пока каждое блюдо не будет полностью приготовлено. По крайней мере, ваша еда будет холодной. В худшем случае ваш официант просто потратил впустую довольно значительное количество времени, казалось бы, без какой-либо веской причины.  
  
В программировании синхронные программы выделяют один поток для обработки запроса или выполнения задачи. Поскольку синхронные операции выполняются по одной, такая кропотливо длинная задача, как запрос к базе данных, тем временем заблокирует все остальные потоки. В то время как асинхронные операции могут выполнять несколько задач одновременно в одном потоке, синхронные программы имеют очередь задач, в которой каждая вторая задача остается бездействующей, пока первая завершена. По определению синхронный означает «подключенный» или «зависимый». По сути, синхронная связь тесно связана. Для реактивных систем, когда программы работают лучше всего, когда они реагируют на входные данные из окружающей среды, могут быть полезны синхронные программы.  
  
**Плюсы синхронного программирования**  
Хотите верьте, хотите нет, но есть веские причины, по которым предприятия и разработчики обращаются к синхронному выполнению, в свою очередь, для асинхронного программирования.  
  
Простота  
Синхронное программирование хорошо поддерживается всеми языками программирования. Разработчикам не нужно беспокоиться о том, можно ли кодировать асинхронные приложения, так как это значение по умолчанию. Кроме того, писать синхронные программы объективно проще, чем асинхронные. Используя блокирующую архитектуру ввода-вывода, вы можете сэкономить время в процессе разработки программного обеспечения и выполнять тесты с меньшими сложностями.  
  
Маркетинговый потенциал  
Поисковым системам легче сканировать веб-страницы, использующие традиционную синхронную архитектуру. Для маркетологов, которые зависят от поисковой оптимизации (SEO) для создания своей репутации и узнаваемости бренда, это заметное преимущество. Чем больше людей просматривают ваш веб-сайт через Google или Bing, тем больше посетителей будет на вашей веб-странице. Естественно, это положительно скажется на вашем возврате инвестиций (ROI).  
  
**Минусы синхронного программирования**Минусы синхронного программирования должны быть довольно очевидны. Это основные недостатки, подталкивающие многих разработчиков к асинхронному программированию.  
  
Скорость  
Время загрузки может быть медленнее при синхронном программировании по сравнению с асинхронным программированием. Этого следовало ожидать, учитывая то, как синхронные программы обрабатывают несколько запросов. Когда поток блокируется, другие потоки в очереди также блокируются. Проще говоря, синхронное программирование похоже на посещение Disney World без VIP-пропуска .  
  
Ресурсоемкость  
Включение синхронного программирования также требует значительных ресурсов. В то время как несколько асинхронных исполнений могут работать в одном потоке, это не относится к синхронным функциям. Вам потребуется больше потоков для обработки большего количества запросов, и чаще всего это оказывается непосильным.  
  
**Когда использовать асинхронное программирование**  
Самый большой вклад, который обеспечивает асинхронное программирование, — это повышение пропускной способности.  
  
Производительность относится к результатам, которые продукт или компания могут произвести в течение определенного времени. Тем не менее, синхронное программирование не является конечной целью всех методологий программирования. Есть определенные сценарии, в которых подходит асинхронное программирование, и другие, в которых асинхронное программирование не подходит. Чтобы узнать, что есть что, вам необходимо определить зависимости и процессы в вашей системе. Параллельные вычисления хорошо подходят для ориентированного на пользователя программирования, но любой другой тип программы не стоит такой сложности. Ограничьте использование асинхронного программирования обработкой задач на основе событий, когда полезно высвобождать ценные ресурсы раньше. Операции ввода-вывода и запросы к базе данных являются распространенными вариантами использования асинхронного программирования.  
  
**Когда использовать синхронное программирование**  
Как известно, компьютеры работают быстро. Таким образом, синхронное программирование занимает не так много времени, как вы можете себе представить. Если вы просто хотите разработать внешнее приложение или выполнить базовую функцию центрального процессора (ЦП), то асинхронное программирование выходит за рамки допустимого. Рендеринг видео или математические вычисления, например, используют центральный процессор для максимальной функциональности. Использование асинхронного программирования для этих типов задач перегрузило бы ЦП и принесло бы больше вреда, чем пользы.  
  
Сверхинжиниринг описывает акт разработки решений проблемы сложным образом. Само собой разумеется, что вы должны избегать чрезмерной инженерии. Стреляйте в синхронное программирование, когда вы можете обойтись.  
  
**Синхронное и асинхронное программирование: что лучше?**  
Между асинхронным и синхронным программированием нет лучшего метода программирования по своей сути. Скорее, ключевым моментом является оценка ваших потребностей в программировании и выбор наиболее оптимального решения для ваших требований к программному обеспечению.

**Проблемы при использовании callback-функций?**

Какие преимущества и недостатки при использовании промисов вместо колбэков (callbacks)?[​](https://www.frontendinterviewhandbook.com/ru/javascript-questions" \l "какие-преимущества-и-недостатки-при-использовании-промисов-вместо-колбэков-callbacks)

**Преимущества**

* Помогает избежать "callback hell", который может быть нечитаемым.
* Упрощает написание последовательного удобочитаемого асинхронного кода с помощью .then().
* Упрощает написание параллельного асинхронного кода с помощью Promise.all().
* С использованием промисов можно избежать следующих проблем, которые возникают при использовании callback-функций:
  + Колбэк-функция была вызвана слишком рано
  + Колбэк-функция была вызвана слишком поздно (или вовсе не была вызвана)
  + Колбэк-функция была вызвана слишком мало или слишком много раз
  + Не удалось передать необходимую среду/параметры
  + Были пропущены ошибки/исключения

**Недостатки**

* Чуть более сложный код (спорно).
* В старых браузерах, где не поддерживается ES2015, вам нужно загрузить полифил, чтобы их использовать.

**Как выполнить несколько асинхронных операций последовательно?**

Промисы придумали, чтобы организовывать асинхронный код последовательно.

Та же последовательность запросов из прошлого примера, но переписанная с использованием промисов.

function request(url) {

return new Promise(function (resolve, reject) {

let responseFromServer

/\*...\*/

resolve(responseFromServer)

})

}

request('/api/users/1')

.then((user) => request(`/api/photos/${user.id}/`))

.then((photo) => request(`/api/crop/${photo.id}/`))

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

<https://doka.guide/js/async-in-js/>

**Какие проблемы может вызвать неправильное использование асинхронности в JavaScript?**

<https://blog.openreplay.com/javascript-async-programming-tips-tricks-and-gotchas/>

JavaScript is an asynchronous language, and handling asynchronous code can be challenging. Common issues with asynchronous code include callback hell, race conditions, and unhandled promise rejections.

**How to identify:** Asynchronous code issues can manifest in various ways, such as incorrect execution order, unhandled errors, or unexpected results. Debugging your code and analyzing the execution flow can help you identify problems with asynchronous operations.

**How to solve:** To overcome asynchronous code issues, use modern techniques like Promises and async/await. These make it easier to handle asynchronous operations and reduce the risk of common pitfalls like callback hell and race conditions. Additionally, always handle errors in your asynchronous code, either by using try/catch blocks with async/await or by attaching error handlers to Promises.

<https://www.dannymoerkerke.com/blog/what-you-need-to-know-about-asynchronous-programming-in-javascript/>

**ECMAScript:**

**Что такое ECMAScript? В чём отличие от JavaScript?**

Есть такой стандарт, который называется ECMA-262. Он создан и разработан организацией ECMA International. В нем прописана спецификация скриптового языка общего назначения. ECMAScript – это спецификация, стандарт скриптового языка программирования. Он содержит правила, сведения и рекомендации, которые должны соблюдаться скриптовым языком, чтобы он считался совместимым с ECMAScript. А JavaScript – это непосредственно язык, соответствующий спецификации ECMAScript. Иными словами – это просто одна из реализаций. А самая революционная версия стандарта – это версия ES6 или версия ES 2015 или ECMAScript 2015, где появились let и const, стрелочные функции, классы, промисы, генераторы и т.д.

ECMAScript (или ES) — это *спецификация*. То есть набор правил и рекомендаций, которые должны соблюдаться языком, чтобы он считался совместимым с этой спецификацией.

💡

JavaScript (или JS) — это *язык*, который соответствует спецификации ECMAScript.

Разница между ними в том, что документация ES говорит, как *создать язык*, который будет соответствовать «семейству» подобных ECMAScript-языков. А документация JS — как *использовать* конкретно этот язык (JS).

[ДОКА](https://doka.guide/js/language-versions/" \l ":~:text=ECMAScript (или ES) — это,считался совместимым с этой спецификацией.&text=JavaScript (или JS) — это,язык%2C который соответствует спецификации ECMAScript.)

[О разных стандартах](https://tproger.ru/translations/wtf-is-ecmascript/)

<https://www.w3schools.com/js/js_versions.asp>

**Разница между let, const и var?**

Ответ в другой части доки о js

**Можно ли изменить значение определённое через const?**

При попытке переопределения const переменной (константы) вызывает ошибку. Но саму константу можно изменить, если это объект. Переназначение объекта константе вызывает ошибку, потому что в саму константу записывается не значение, а ссылка на объект. Сам же объект хранится в области оперативной памяти, где мы можем изменять все, что угодно, сохраняя при этом ссылку, которую использует константа.



**Что такое временная мёртвая зона (temporal dead zone)?**

В стандарте ECMAScript 2015 переменные, объявленные оператором let, переносятся в начало блока. Но если вы сошлетесь в блоке на переменную, до того как она объявлена оператором let, то это приведет к выбросу исключения ReferenceError, потому что переменная находится во “временной мертвой зоне” с начала блока и до места ее объявления. (В отличии от переменной, объявленной через var, которая просто будет содержать значение undefined)

Большинство ресурсов упускают важную деталь при определении Temporal Dead Zone. Переменная все так же попадает в замыкания даже до момента ее инициализиции, ее видят функции. ReferenceError бросается когда имеет место попытка обращения к переменной раньше ее инициализации, но речь именно об инструкциях обращения и их порядке, а не о расположении обьявления переменнойы в коде.

Если говорить об инструкциях на примере выше, то движок будет обрабатывать код следующим образом (на пальцах):

сперва строит LexicalEnvironment функции в promise;

на этапе построения создаются переменные объявленные с использованием ключевых слов let и const, но не происходит инцииализации переменных;

начинается выполнение инструкций;

если инструкция пытается обратиться к переменной объявленной с let или const, но которая еще неинициализирована, то произойдет бросок ReferenceError;

при встрече инструкции по созданию функции onResolve создастся LexicalEnvironment ассоциированный с этой функцией и привязывается LexicalEnvironment из первого пункта, другими словами, создается замыкание в которые попали переменные onReject, emitter, resolve, reject, etc

обращения к onReject не происходило, потому никаких исключений нет

к моменту когда начнет выполняться onResolve переменная onReject уже будет инициирована и обращение не вызовет исключения

<https://dmenshikov.com/2017-07-13-temporal-dead-zone/>

**Временная мертвая зона**

Попытавшись обратиться к let или const-переменной до ее объявления, мы получаем ошибку, и происходит это из-за *временной мертвой зоны* (temporal dead zone, TDZ).

TDZ начинается в начале области видимости переменной и заканчивается ее объявлением. Обращение к переменной в TDZ приводит к выбросу ReferenceError.

Вот пример понятного блока, показывающего начало и конец временной мертвой зоны foo:

**{**

// Начало TDZ для foo

**let** bar = 'bar';

console.log**(**bar**)**; // "bar"

console.log**(**foo**)**; // ReferenceError, потому что мы в TDZ

**let** foo = 'foo'; // Конец TDZ для foo

**}**

TDZ также присутствует в дефолтных параметрах функции, которые оцениваются слева направо. В следующем примере bar находится в TDZ, пока не получает значение по умолчанию:

**function** foobar**(**foo = bar, bar = 'bar'**)** **{**

console.log**(**foo**)**;

**}**

foobar**()**; // Uncaught ReferenceError: Cannot access 'bar' before initialization

Но этот код работает, потому что мы можем обратиться к foo вне ее TDZ:

**function** foobar**(**foo = 'foo', bar = foo**)** **{**

console.log**(**bar**)**;

**}**

foobar**()**; // "foo"

**typeof во временной мертвой зоне**

Если вы используете let или const-переменную как операнд оператора typeof в TDZ, вы получите ошибку:

console.log**(**typeof foo**)**; // Uncaught ReferenceError: Cannot access 'foo' before initialization

**let** foo = 'foo';

Это поведение согласуется с другими примерами с let и const в TDZ, которые мы уже видели. Здесь мы получаем ReferenceError, потому что foo объявлена, но не инициализирована. Мы должны знать, что используем ее до инициализации.

Но это не касается использования var-переменной до ее объявления, потому что такая переменная при поднятии инициализируется со значением undefined:

console.log**(**typeof foo**)**; // "undefined"

**var** foo = 'foo';

Кроме того (и это удивительно), мы можем проверить тип несуществующей переменной и не получить ошибку. typeof безопасно возвращает строку:

console.log**(**typeof foo**)**; // "undefined"

Фактически, появление let и const сломало гарантию того, что typeof всегда возвращает строковое значение для любого операнда.

<https://techrocks.ru/2021/11/30/hoisting-in-javascript/>

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#temporal\_dead\_zone\_tdz](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let%23temporal_dead_zone_tdz)

**Разница между Rest и Spread операторами?**

Когда мы видим "..." в коде, это могут быть как остаточные параметры, так и оператор расширения.

Как отличить их друг от друга:

* Если ... располагается в конце списка параметров функции, то это «остаточные параметры». Он собирает остальные неуказанные аргументы и делает из них массив.
* Если ... встретился в вызове функции или где-либо ещё, то это «оператор расширения». Он извлекает элементы из массива.

Полезно запомнить:

* Остаточные параметры используются, чтобы создавать новые функции с неопределённым числом аргументов.
* С помощью оператора расширения можно вставить массив в функцию, которая по умолчанию работает с обычным списком аргументов.

Вместе эти конструкции помогают легко преобразовывать наборы значений в массивы и обратно.

К аргументам функции можно обращаться и по-старому — через псевдомассив arguments.

<https://learn.javascript.ru/rest-parameters-spread-operator>

The spread operator spreads the values in an array or a string across one or more arguments. Rest operator allows us to pass an indefinite number of arguments to function by accumulating these several values into an array.

<https://www.freecodecamp.org/news/javascript-rest-vs-spread-operators/>

In Javascript, both the spread operator and rest parameter have the same syntax which is three dots(…). Even though they have the same syntax they differ in functions.

Spread operator: The spread operator helps us expand an iterable such as an array where multiple arguments are needed, it also helps to expand the object expressions. In cases where we require all the elements of an iterable or object to help us achieve a task, we use a spread operator.

Rest operator: The rest parameter is converse to the spread operator. while the spread operator expands elements of an iterable, the rest operator compresses them. It collects several elements. In functions when we require to pass arguments but were not sure how many we have to pass, the rest parameter makes it easier.

**Что такое деструктуризация?**

В JavaScript есть две чаще всего используемые структуры данных – это Object и Array.

* Объекты позволяют нам создавать одну сущность, которая хранит элементы данных по ключам.
* Массивы позволяют нам собирать элементы данных в упорядоченный список.

Но когда мы передаём их в функцию, то ей может понадобиться не объект/массив целиком, а элементы по отдельности.

*Деструктурирующее присваивание* – это специальный синтаксис, который позволяет нам «распаковать» массивы или объекты в несколько переменных, так как иногда они более удобны.

Деструктуризация также прекрасно работает со сложными функциями, которые имеют много параметров, значений по умолчанию и так далее. Скоро мы увидим это.

**«Деструктурирующее присваивание» не уничтожает массив. Оно вообще ничего не делает с правой частью присваивания, его задача – только скопировать нужные значения в переменные.**

[**https://learn.javascript.ru/destructuring-assignment**](https://learn.javascript.ru/destructuring-assignment)

Выражения объявления объектов или массивов предоставляют простой способ создания пакета однородных данных. При создании такого пакета вы получаете возможность использовать его любым доступным образом. Также вы можете возвращать его в функциях.

Одной из ключевых возможностей использования деструктурирующего присваивания является чтение структуры данных одним оператором, хотя помимо этого вы можете найти множество других применений в приведённых ниже примерах.

[Еще информация](https://ru.hexlet.io/courses/js-objects/lessons/destructuring/theory_unit" \l ":~:text=Деструктуризация (destructuring) — специальный синтаксис,и сделать его более понятным.)

**Для чего используется цикл for…of?**

Оператор for...of выполняет цикл обхода [итерируемых объектов (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols) (включая [Array](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array), [Map (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map), [Set](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Set), объект [аргументов (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments) и подобных), вызывая на каждом шаге итерации операторы для каждого значения из различных свойств объекта.

**[Синтаксис](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/for...of" \l "синтаксис)**

for (variable of iterable) {

statement

}

variable

На каждом шаге итерации variable присваивается значение нового свойства объекта *iterable*. Переменная *variable* может быть также объявлена с помощью const, let или var.

iterable

Объект, перечисляемые свойства которого обходятся во время выполнения цикла.

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/for...of>

**for ... of** цикл работает только с итерируемыми объектами. В JavaScript итерируемые объекты - это объекты, которые можно перебирать.

String, Array, TypedArray, Map и Set - все это встроенные итерируемые объекты, поскольку каждый из их объектов-прототипов реализует метод @@iterator. Итак, цикл for ... of работает с указанными типами объектов.

Например:

cosnt str = "Hello World";

for(element of str) {

console.log(element);

}

// H e l l o " " W o r l d

Скопировать код

for...in Loop

[The MDN Definition](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in):

**Цикл for...in** проходит через перечисляемые свойства объекта. Он пройдёт по каждому отдельному элементу.

for ... in работает с теми свойствами, у которых enumerable flag установлен в true:

* Перечислимый флаг для свойств, созданных с помощью простого присваивания или инициализатора свойств, по умолчанию равен true.
* Перечислимый флаг для свойств, созданных с помощью Object.defineProperty, по умолчанию равен false.
* Также работает со строками и массивами, поскольку перечислимый флаг для свойств строки и массива также по умолчанию имеет значение true.

Например:

const student = {

name: "Ivan",

age: 35,

}

for(key in student) {

console.log(key, student[key]);

}

// result:

// name: "ivan", age: 33

Скопировать код

Что использовать for...of или for...in

В заключение, что же использовать:

* **for...of** - Используется для перебора строк и массивов.
* **for...in** - Используйте для перебора объектов.

От себя добавлю, что **for...of** поставляется с ES2015 и в настоящее время является лучшей практикой (по моему опыту) потому что:

* Более читаемый,
* Без обратных вызовов,
* Быстрее (если верить тестам),
* Вы также можете использовать .entries() и деструктурирование.

Do not use **for in** over an Array if the index **order** is important.

The index order is implementation-dependent, and array values may not be accessed in the order you expect.

It is better to use a **for** loop, a **for of** loop, or **Array.forEach()** when the order is important.

**Что такое шаблонные литералы (Template Literals)?**

Шаблонными литералами называются строковые литералы, допускающие использование выражений внутри. С ними вы можете использовать многострочные литералы и строковую интерполяцию. В спецификациях до ES2015 они назывались "шаблонными строками".

**[Синтаксис](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals" \l "синтаксис)**

`строка текста`

`строка текста 1

строка текста 2`

`строка текста ${выражение} строка текста`

tag `строка текста ${выражение} строка текста`

**[Описание](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals" \l "описание)**

Шаблонные литералы заключены в обратные кавычки (` `) вместо двойных или одинарных. Они могут содержать подстановки, обозначаемые знаком доллара и фигурными скобками (${выражение}). Выражения в подстановках и текст между ними передаются в функцию. По умолчанию функция просто объединяет все части в строку. Если перед строкой есть выражение (здесь это tag), то шаблонная строка называется "теговым шаблоном". В этом случае, теговое выражение (обычно функция) вызывается с обработанным шаблонным литералом, который вы можете изменить перед выводом. Для экранирования обратной кавычки в шаблонных литералах указывается обратный слеш **\**.

`\`` === '`' // --> true

Copy to Clipboard

**[Многострочные литералы](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals" \l "многострочные_литералы)**

Символы новой строки являются частью шаблонных литералов. Используя обычные строки, вставка переноса потребовала бы следующего синтаксиса:

console.log('string text line 1\n' +

'string text line 2');

// "string text line 1

// string text line 2"

Copy to Clipboard

То же с использованием шаблонных литералов:

console.log(`string text line 1

string text line 2`);

// "string text line 1

// string text line 2"

Copy to Clipboard

**[Интерполяция выражений](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals" \l "интерполяция_выражений)**

Для вставки выражений в обычные строки вам пришлось бы использовать следующий синтаксис:

var a = 5;

var b = 10;

console.log('Fifteen is ' + (a + b) + ' and not ' + (2 \* a + b) + '.');

// "Fifteen is 15 and not 20."

Copy to Clipboard

Теперь, при помощи шаблонных литералов, вам доступен "синтаксический сахар", делающий подстановки вроде той более читабельными:

var a = 5;

var b = 10;

console.log(`Fifteen is ${a + b} and not ${2 \* a + b}.`);

// "Fifteen is 15 and not 20."

Copy to Clipboard

**[Вложенные шаблоны](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals" \l "вложенные_шаблоны)**

Временами, вложить шаблон — это кратчайший и, возможно, более читабельный способ составить строку. Просто поместите внутрь шаблона с обратными кавычками ещё одни, обернув их в подстановку ${ }. Например, если выражение истинно, можно вернуть шаблонный литерал.

В ES5:

var classes = 'header'

classes += (isLargeScreen() ?

'' : item.isCollapsed ?

' icon-expander' : ' icon-collapser');

Copy to Clipboard

В ES2015 с шаблонными литералами без вложения:

const classes = `header ${ isLargeScreen() ? '' :

(item.isCollapsed ? 'icon-expander' : 'icon-collapser') }`;

Copy to Clipboard

В ES2015 с вложенными шаблонными литералами:

const classes = `header ${ isLargeScreen() ? '' :

`icon-${item.isCollapsed ? 'expander' : 'collapser'}` }`;

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

<https://habr.com/ru/companies/ruvds/articles/511590/>

**Что такое Set, Map, WeakMap и WeakSet?**

* Map – коллекция записей вида ключ: значение, лучше Object тем, что перебирает всегда в порядке вставки и допускает любые ключи.
* Set – коллекция уникальных элементов, также допускает любые ключи.

Основная область применения Map – ситуации, когда строковых ключей не хватает (нужно хранить соответствия для ключей-объектов), либо когда строковый ключ может быть совершенно произвольным.

К примеру, в обычном объекте Object нельзя использовать «совершенно любые» ключи. Есть встроенные методы, и уж точно есть свойство с названием \_\_proto\_\_, которое зарезервировано системой. Если название ключа даётся посетителем сайта, то он может попытаться использовать такое свойство, заменить прототип, а это, при запуске JavaScript на сервере, уже может привести к серьёзным ошибкам.

* WeakMap и WeakSet – «урезанные» по функциональности варианты Map/Set, которые позволяют только «точечно» обращаться к элементам (по конкретному ключу или значению). Они не препятствуют сборке мусора, то есть, если ссылка на объект осталась только в WeakSet/WeakMap – она будет удалена.

<https://learn.javascript.ru/set-map>

<https://learn.javascript.ru/map-set>

<https://learn.javascript.ru/weakmap-weakset>

[Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map) – коллекция пар ключ-значение.

Методы и свойства:

* [new Map([iterable])](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/Map) – создаёт коллекцию, можно указать перебираемый объект (обычно массив) из пар [ключ,значение] для инициализации.
* [map.set(key, value)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/set) – записывает по ключу key значение value.
* [map.get(key)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/get) – возвращает значение по ключу или undefined, если ключ key отсутствует.
* [map.has(key)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/has) – возвращает true, если ключ key присутствует в коллекции, иначе false.
* [map.delete(key)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/delete) – удаляет элемент по ключу key.
* [map.clear()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/clear) – очищает коллекцию от всех элементов.
* [map.size](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/size) – возвращает текущее количество элементов.

Отличия от обычного объекта Object:

* Что угодно может быть ключом, в том числе и объекты.
* Есть дополнительные методы, свойство size.

[Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set) – коллекция уникальных значений, так называемое «множество».

Методы и свойства:

* [new Set(iterable)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/Set) – создаёт Set, можно указать перебираемый объект со значениями для инициализации.
* [set.add(value)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/add) – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект set.
* [set.delete(value)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/delete) – удаляет значение, возвращает true если value было в множестве на момент вызова, иначе false.
* [set.has(value)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/has) – возвращает true, если значение присутствует в множестве, иначе false.
* [set.clear()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/clear) – удаляет все имеющиеся значения.
* [set.size](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/size) – возвращает количество элементов в множестве.

Перебор Map и Set всегда осуществляется в порядке добавления элементов, так что нельзя сказать, что это – неупорядоченные коллекции, но поменять порядок элементов или получить элемент напрямую по его номеру нельзя.

WeakMap – это Map-подобная коллекция, позволяющая использовать в качестве ключей только объекты, и автоматически удаляющая их вместе с соответствующими значениями, как только они становятся недостижимыми иными путями.

WeakSet – это Set-подобная коллекция, которая хранит только объекты и удаляет их, как только они становятся недостижимыми иными путями.

Обе этих структуры данных не поддерживают методы и свойства, работающие со всем содержимым сразу или возвращающие информацию о размере коллекции. Возможны только операции на отдельном элементе коллекции.

WeakMap и WeakSet используются как вспомогательные структуры данных в дополнение к «основному» месту хранения объекта. Если объект удаляется из основного хранилища и нигде не используется, кроме как в качестве ключа в WeakMap или в WeakSet, то он будет удалён автоматически.

**Разница между обычными функциями и стрелочными?**

<https://techrocks.ru/2021/08/28/regular-and-arrow-functions-in-javascript/>

Значение *this* внутри обыкновенной функции динамически зависит от контекста вызова. Собственный *this* внутри стрелочной функции отсутствует и она ссылается на *this* внешней функции. Массив *arguments* внутри обыкновенной функции содержит список аргументов функции. Стрелочная функция, не имеет массива *arguments*(но ты можешь использовать деструктуризацию, для иммитации аналога *...args*).Если в стрелочной функции содержится одна инструкция, то ты можешь использовать неявный *return*, даже без использования ключевого слова *return*. Последнее в списке, но не по важности - ты можешь использовать синтаксис стрелочных функций для внутри класса. При этом в качестве *this*будет выступать объект класса.

Стрелочные функции отличаются от обычных не только способом записи. Главное их отличие проявляется в том, как они работают с контекстом. Вкратце: контекст обычных функций зависит от места вызова, а контекст стрелочных функций — от того места, где они были определены.

Стрелочные функции:

* Не имеют this.
* Не имеют arguments.
* Не могут быть вызваны с new.
* (У них также нет super

<https://learn.javascript.ru/arrow-functions>

Стрелочная функция — это относительно новый способ создания функций в JS. Стрелочные функции создаются быстрее и имеют более читаемый синтаксис, чем функциональные выражения. В стрелочных функциях опускается слово «function»:

// ES5

**var** getCurrentDate = **function**(){

**return** **new** Date()

}

// ES6

**const** getCurrentDate = () => **new** Date()

В функциональном выражении мы используем ключевое слово «return» для возврата значения. В стрелочной функции мы этого не делаем, поскольку стрелочные функции неявно возвращают значения при условии, что мы возвращаем одно выражение или значение:

// ES5

**function** **greet**(name){

**return** 'Hello ' + name + '!'

}

// ES6

**const** greet = (name) => `Hello ${name}`

**const** greet2 = name = > `Hello ${name}`

Мы также можем передавать параметры стрелочным функциям. Если мы передаем один параметр, его можно не оборачивать в круглые скобки:

**const** getArgs = () => arguments

**const** getArgs2 = (...rest) => rest

У стрелочных функций нет доступа к объекту arguments. Поэтому вызов первой функции приведет к ошибке. Для получения параметров, переданных функции, мы можем использовать оператор rest.

**const** data = {

result: 0

nums: [1,2,3,4,5]

**computeResult**(){

// this ссылается на объект data

**const** addAll = () => {

// стрелочные функции копируют значение this из лексического окружения

**return** this.nums.reduce((total, cur) => total + cur, 0)

}

this.result = addAll()

}

}

**Разница между методом Object.freeze() и const?**

const применяется только для неизменяемой ссылки на ячейку памяти со значением, что означает невозможность задать новое значение для переменной. Object. freeze работает со значениями объектов. Делает объект неизменяемым, то есть изменить его свойства невозможно.

**Что такое итераторы?**

Объекты, которые можно использовать в цикле for..of, называются *итерируемыми*.

* Технически итерируемые объекты должны иметь метод Symbol.iterator.
  + Результат вызова obj[Symbol.iterator] называется *итератором*. Он управляет процессом итерации.
  + Итератор должен иметь метод next(), который возвращает объект {done: Boolean, value: any}, где done:true сигнализирует об окончании процесса итерации, в противном случае value – следующее значение.
* Метод Symbol.iterator автоматически вызывается циклом for..of, но можно вызвать его и напрямую.
* Встроенные итерируемые объекты, такие как строки или массивы, также реализуют метод Symbol.iterator.
* Строковый итератор знает про суррогатные пары.

Объекты, имеющие индексированные свойства и length, называются *псевдомассивами*. Они также могут иметь другие свойства и методы, но у них нет встроенных методов массивов.

Если мы заглянем в спецификацию, мы увидим, что большинство встроенных методов рассчитывают на то, что они будут работать с итерируемыми объектами или псевдомассивами вместо «настоящих» массивов, потому что эти объекты более абстрактны.

Array.from(obj[, mapFn, thisArg]) создаёт настоящий Array из итерируемого объекта или псевдомассива obj, и затем мы можем применять к нему методы массивов. Необязательные аргументы mapFn и thisArg позволяют применять функцию с задаваемым контекстом к каждому элементу.

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

Объект является итератором, если он умеет обращаться к элементам коллекции по одному за раз, при этом отслеживая своё текущее положение внутри этой последовательности. В JavaScript итератор - это объект, который предоставляет метод next(), возвращающий следующий элемент последовательности. Этот метод возвращает объект с двумя свойствами: done и value.

После создания, объект-итератор может быть явно использован, с помощью вызовов метода next().

function makeIterator(array){

var nextIndex = 0;

return {

next: function(){

return nextIndex < array.length ?

{value: array[nextIndex++], done: false} :

{done: true};

}

}

}

Copy to Clipboard

После инициализации, метод next() может быть вызван для поочерёдного доступа к парам ключ-значение в объекте:

var it = makeIterator(['yo', 'ya']);

console.log(it.next().value); // 'yo'

console.log(it.next().value); // 'ya'

console.log(it.next().done); // true

**[Итерируемые объекты](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Iterators_and_generators" \l "итерируемые_объекты)**

Объект является итерируемым, если в нем определён способ перебора значений, то есть, например, как значения перебираются в конструкции [for..of](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/for...of). Некоторые встроенные типы, такие как [Array](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array) или [Map (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map), по умолчанию являются итерируемыми, в то время как другие типы, как, например, [Object](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object), таковыми не являются.

Чтобы быть итерируемым, объект обязан реализовать метод **@@iterator**, что означает, что он (или один из объектов выше по [цепочке прототипов](https://developer.mozilla.org/ru/docs/Web/JavaScript/Inheritance_and_the_prototype_chain)) обязан иметь свойство с именем [Symbol.iterator](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Symbol/iterator):

**[Пользовательские итерируемые объекты](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Iterators_and_generators" \l "пользовательские_итерируемые_объекты)**

Мы можем создать свои собственные итерируемые объекты так:

var myIterable = {}

myIterable[Symbol.iterator] = function\* () {

yield 1;

yield 2;

yield 3;

};

[...myIterable] // [1, 2, 3]

Copy to Clipboard

**[Встроенные итерируемые объекты](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Iterators_and_generators" \l "встроенные_итерируемые_объекты)**

Объекты [String](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String), [Array](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array), [TypedArray](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/TypedArray), [Map (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map) и [Set](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Set) являются итерируемыми, потому что их прототипы содержат метод [Symbol.iterator](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Symbol/iterator).

**[Синтаксис для итерируемых объектов](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Iterators_and_generators" \l "синтаксис_для_итерируемых_объектов)**

Некоторые выражения работают с итерируемыми объектами, например, [for-of](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of) циклы, [spread operator](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Spread_syntax), [yield\*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield*), и [destructuring assignment](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment).

for(let value of ["a", "b", "c"]){

console.log(value)

}

// "a"

// "b"

// "c"

[..."abc"] // ["a", "b", "c"]

function\* gen(){

yield\* ["a", "b", "c"]

}

gen().next() // { value:"a", done:false }

[a, b, c] = new Set(["a", "b", "c"])

a // "a"

**Что такое генераторы? Когда стоит использовать генераторы?**

В то время как пользовательские итераторы могут быть весьма полезны, при их программировании требуется уделять серьёзное внимание поддержке внутреннего состояния. [**Генераторы**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Generator) предоставляют мощную альтернативу: они позволяют определить алгоритм перебора, написав единственную функцию, которая умеет поддерживать собственное состояние.

Генераторы - это специальный тип функции, который работает как фабрика итераторов. Функция становится генератором, если содержит один или более [yield](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/yield) операторов и использует [function\*](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/function*) синтаксис.

function\* idMaker(){

var index = 0;

while(true)

yield index++;

}

var it = idMaker();

console.log(it.next().value); // 0

console.log(it.next().value); // 1

console.log(it.next().value); // 2

// ...

Генераторы вычисляют результаты своих yield выражений по требованию, что позволяет им эффективно работать с последовательностями с высокой вычислительной сложностью, или даже с бесконечными последовательностями, как продемонстрировано выше.

Метод [next()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Generator/next) также принимает значение, которое может использоваться для изменения внутреннего состояния генератора. Значение, переданное в next(), будет рассматриваться как результат последнего yield выражения, которое приостановило генератор.

Вот генератор чисел Фибоначчи, использующий next(x) для перезапуска последовательности:

function\* fibonacci(){

var fn1 = 1;

var fn2 = 1;

while (true){

var current = fn2;

fn2 = fn1;

fn1 = fn1 + current;

var reset = yield current;

if (reset){

fn1 = 1;

fn2 = 1;

}

}

}

var sequence = fibonacci();

console.log(sequence.next().value); // 1

console.log(sequence.next().value); // 1

console.log(sequence.next().value); // 2

console.log(sequence.next().value); // 3

console.log(sequence.next().value); // 5

console.log(sequence.next().value); // 8

console.log(sequence.next().value); // 13

console.log(sequence.next(true).value); // 1

console.log(sequence.next().value); // 1

console.log(sequence.next().value); // 2

console.log(sequence.next().value); // 3

Copy to Clipboard

**Примечание:** Интересно, что вызов next(undefined) равносилен вызову next(). При этом вызов next() для нового генератора с любым аргументом, кроме undefined, спровоцирует исключение TypeError.

Можно заставить генератор выбросить исключение, вызвав его метод [throw()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Generator/throw) и передав в качестве параметра значение исключения, которое должно быть выброшено. Это исключение будет выброшено из текущего приостановленного контекста генератора так, будто текущий приостановленный yield оператор являлся throw оператором.

Если yield оператор не встречается во время обработки выброшенного исключения, то исключение передаётся выше через вызов throw(), и результатом последующих вызовов next() будет свойство done равное true.

У генераторов есть метод [return(value)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Generator/return), который возвращает заданное значение и останавливает работу генератора.

* Генераторы создаются при помощи функций-генераторов function\* f(…) {…}.
* Внутри генераторов и только внутри них существует оператор yield.
* Внешний код и генератор обмениваются промежуточными результатами посредством вызовов next/yield.

В современном JavaScript генераторы используются редко. Но иногда они оказываются полезными, потому что способность функции обмениваться данными с вызывающим кодом во время выполнения совершенно уникальна. И, конечно, для создания перебираемых объектов.

Также, в следующей главе мы будем изучать асинхронные генераторы, которые используются, чтобы читать потоки асинхронно сгенерированных данных (например, постранично загружаемые из сети) в цикле for await ... of.

В веб-программировании мы часто работаем с потоками данных, так что это ещё один важный случай использования.

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

<https://learn.javascript.ru/async-iterators-generators>

<https://habr.com/ru/companies/ruvds/articles/417481/>

[Medium](https://nickbulljs.medium.com/зачем-они-нужны-в-javascript-symbol-iterator-generator-d5d186b4f1bd" \l ":~:text=Это функция которая может приостанавливать,Так выглядят генераторы.&text=Когда мы вызываем функцию-генератор,она возвращает нам объект-итератор.)

**Что такое ES6 модули?**

Модули позволяют объединять (использовать) код из разных файлов и избавляют нас от необходимости держать весь код в одном большом файле. До появления модулей в JS существовало две популярные системы модулей для поддержки кода:

* CommonJS — Nodejs
* AMD (AsyncronousModuleDefinition) — Browsers

Синтаксис модулей очень простой: мы используем import для импорта функциональности или значений из другого файла или файлов и export для экспорта.  
  
Экспорт функциональности в другой файл (именной экспорт):

// ES5 CommonJS - helpers.js

exports.isNull = **function**(val){

**return** val === null

}

exports.isUndefined = **function**(val){

**return** val === undefined

}

exports.isNullOrUndefined = **function**(val){

**return** exports.isNull(val) || exports.isUndefined(val)

}

// ES6 модули

**export** **function** **isNull**(val){

**return** val === null;

}

**export** **function** **isUndefined**(val) {

**return** val === undefined;

}

**export** **function** **isNullOrUndefined**(val) {

**return** isNull(val) || isUndefined(val);

}

Импорт функциональности в другой файл:

// ES5 CommonJS - index.js

**const** helpers = require('./helpers.js')

**const** isNull = helpers.isNull

**const** isUndefined = helpers.isUndefined

**const** isNullOrUndefined = helpers.isNullOrUndefined

// либо с помощью деструктуризации

**const** { isNull, isUndefined, isNullOrUndefined } = require('./helpers.js')

// ES6 модули

**import** \* **as** helpers **from** './helpers.js' // helpers - это объект

// либо

**import** { isNull, isUndefined, isNullOrUndefined **as** isValid} **from** './helpers.js' // используем "as" для переименовывания

Экспорт по умолчанию:

// ES5 CommonJS - index.js

**class** **Helpers** {

**static** **isNull**(val){

**return** val === null

}

**static** **isUndefined**(val){

**return** val === undefined

}

**static** **isNullOrUndefined**(val){

**return** this.isNull(val) || this.isUndefined(val)

}

}

module.exports = Helpers

// ES6 модули

**class** **Helpers** {

**static** **isNull**(val){

**return** val === null

}

**static** **isUndefined**(val){

**return** val === undefined

}

**static** **isNullOrUndefined**(val){

**return** this.isNull(val) || this.isUndefined(val)

}

}

**export** **default** Helpers

Импорт:

// ES5 CommonJS - index.js

**const** Helpers = require('./helpers.js')

console.log(Helpers.isNull(null))

// ES6 модули

**import** Helpers **from** './helpers.js'

console.log(Helpers.isNull(null))

Это базовое использование модулей. Я не стал вдаваться в подробности, поскольку мой пост и без того получается слишком большим.

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

<https://www.w3schools.com/js/js_modules.asp>

**Что такое символ (Symbol) в ES6?**

Символ (symbol) – примитивный тип данных, использующийся для создания уникальных идентификаторов.

Символы создаются вызовом функции Symbol(), в которую можно передать описание (имя) символа.

Даже если символы имеют одно и то же имя, это – разные символы. Если мы хотим, чтобы одноимённые символы были равны, то следует использовать глобальный реестр: вызов Symbol.for(key) возвращает (или создаёт) глобальный символ с key в качестве имени. Многократные вызовы команды Symbol.for с одним и тем же аргументом возвращают один и тот же символ.

Символы имеют два основных варианта использования:

1. «Скрытые» свойства объектов.

Если мы хотим добавить свойство в объект, который «принадлежит» другому скрипту или библиотеке, мы можем создать символ и использовать его в качестве ключа. Символьное свойство не появится в for..in, так что оно не будет нечаянно обработано вместе с другими. Также оно не будет модифицировано прямым обращением, так как другой скрипт не знает о нашем символе. Таким образом, свойство будет защищено от случайной перезаписи или использования.

Так что, используя символьные свойства, мы можем спрятать что-то нужное нам, но что другие видеть не должны.

1. Существует множество системных символов, используемых внутри JavaScript, доступных как Symbol.\*. Мы можем использовать их, чтобы изменять встроенное поведение ряда объектов. Например, в дальнейших главах мы будем использовать Symbol.iterator для [итераторов](https://learn.javascript.ru/iterable), Symbol.toPrimitive для настройки [преобразования объектов в примитивы](https://learn.javascript.ru/object-toprimitive) и так далее.

Технически символы скрыты не на 100%. Существует встроенный метод [Object.getOwnPropertySymbols(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertySymbols) – с его помощью можно получить все свойства объекта с ключами-символами. Также существует метод [Reflect.ownKeys(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Reflect/ownKeys), который возвращает *все* ключи объекта, включая символьные. Так что они не совсем спрятаны. Но большинство библиотек, встроенных методов и синтаксических конструкций не используют эти методы.

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

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

**Для чего используется метод .includes()?**

Метод includes() определяет, содержит ли массив определённый элемент, возвращая в зависимости от этого true или false

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

[DOKA](https://doka.guide/js/includes/)

**Для чего используется метод .getOwnPropertyDescriptors()?**

getOwnPropertyDescriptors() возвращает все собственные дескрипторы свойств данного объекта.

Этот метод позволяет изучить точное описание всех собственных свойств объекта. Свойство в JavaScript состоит из строкового имени или [Symbol](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Symbol) и свойства дескриптора. Дополнительную информацию о типах свойств дескрипторов и их атрибутах можно найти в [Object.defineProperty()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty).

Свойство дескриптора это запись с некоторыми из следующих атрибутов:

value

Значение, связанное со свойством (только дескрипторы данных).

**writable**

true тогда и только тогда когда значение, связанное со свойством, может быть изменено (только дескрипторы данных).

get

Функция, которая служит в качестве получателя для свойства, или [undefined](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/undefined) если нет получателя (только дескрипторы доступа).

set

Функция, которая служит установщиком для свойства, или [undefined](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/undefined) если установщика нет (только дескрипторы доступа).

configurable

true тогда и только тогда, когда тип этого свойства дескриптора может быть изменён, и если свойство может быть удалено из соответствующего объекта.

enumerable

true тогда и только тогда, когда это свойство отображается при перечислении свойств соответствующего объекта.

**[Примеры](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptors" \l "примеры)**

**[Создание поверхностного клона](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptors" \l "создание_поверхностного_клона)**

В то время как метод [Object.assign()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/assign) будет только копировать перечисляемые и собственные свойства из исходного объекта в целевой объект, вы можете использовать этот метод и [Object.create()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/create) для поверхностного копирования между двумя неизвестными объектами:

Object.create(

Object.getPrototypeOf(obj),

Object.getOwnPropertyDescriptors(obj)

);

<https://learn.javascript.ru/property-descriptors>

<https://learn.javascript.ru/prototype-methods>

<https://doka.guide/js/descriptors/>

JavaScript метод getOwnPropertyDescriptor() позволяет возвратить объект дескриптор для указанного собственного (неунаследованного) свойства заданного объекта. Дескрипторы свойств – это обычные объекты JavaScript, которые описывают атрибуты и значение свойства.

Подробную информацию о дескрипторах свойств вы можете получить в описании метода defineProperty(), который позволяет определить новое или изменить существующее свойство объекта, описывая его дескрипторами.

Для того чтобы возвратить объект, содержащий дескрипторы всех собственных (неунаследованных) свойств заданного объекта, вы можете воспользоваться методом getOwnPropertyDescriptors()ECMAScript 2017.

**Для чего используется метод .fromEntries()?**

fromEntries() Метод Object. fromEntries() преобразует список пар ключ-значение в объект.

Метод Object.fromEntries() принимает список пар ключ-значение и возвращает новый объект, свойства которого задаются этими записями. Ожидается, что аргумент iterable будет объектом, который реализует метод @@iterator, который возвращает объект итератора, который создаёт двухэлементный массивоподобный объект, первый элемент которого является значением, которое будет использоваться в качестве ключа свойства, а второй элемент — значением связанного с этим ключом свойства.

Object.fromEntries() выполняет процедуру, обратную [Object.entries()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/entries).

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

<https://learn.javascript.ru/keys-values-entries>

**Для чего используются методы .flat() и .flatMap()?**

Метод flat() делает входной массив плоским, глубина указывается в аргументах. Метод flatMap() применяет функцию, трансформируя массив и делает массив плоским.

Метод Array.prototype.flat() позволяет работать с вложенными массивами. Обратите внимание на код.

**const** array **=** [1, [2, [3]]];

Здесь есть массив array, в который вложен ещё один массив, в котором в свою очередь находится ещё один массив.

Метод Array.prototype.flat() достаёт вложенные массивы и переносит их на уровень, который определяет разработчик. Дефолтный уровень — 1.

array.flat();

*// [1, 2, [3]]*

*// равнозначно:*

array.flat(1);

*// [1, 2, [3]]*

Если глубина вложенности неизвестна, достать вложенные массивы можно с помощью аргумента Infinity. В этом случае метод [рекурсивно](https://ru.hexlet.io/courses/introduction_to_programming/lessons/recursion/theory_unit) обходит массив и достает все вложенные элементы независимо от глубины.

*// Достаём все вложенные массивы:*

array.flat(**Infinity**);

*// [1, 2, 3]*

Метод Array.prototype.flatMap()

В качестве примера рассмотрим функцию duplicate, которая принимает значение и возвращает массив из двух элементов, каждый из которых повторяет значение. Если применить функцию duplicate к массиву, получим вложенный массив.

**const** duplicate **=** x **=>** [x, x];

[2, 3, 4].map(duplicate);

*// [[2, 2], [3, 3], [4, 4]]*

Извлечь вложенные массивы можно с помощью flat.

[2, 3, 4].map(duplicate).flat();

*// [2, 2, 3, 3, 4, 4]*

В функциональном программировании подобные операции с массивами встречаются часто. Поэтому в ECMAScript появился метод .flatMap(), который упрощает код.

[2, 3, 4].flatMap(duplicate);

*// [2, 2, 3, 3, 4, 4]*

Использовать .flatMap() для преобразования массива эффективнее, чем последовательно использовать .flat() и .map(). Поддержку методов Array.prototype.flat() и Array.prototype.flatMap() в браузерах можно проверить [по ссылке](https://caniuse.com/" \l "search=Array.prototype.flat).

[DOKA](https://doka.guide/js/array-flatmap/" \l ":~:text=Метод flat() делает входной массив,массив и делает массив плоским.)

**Для чего используются методы .padStart() и .padEnd()?**

Методы padStart() и padEnd() растянуть строку на определенное количество символов и заполнить строку слева и справа соответственно.

let hello = "hello".padStart(8); // " hello"

console.log(hello);

hello = "hello".padEnd(8); // "hello "

console.log(hello);

Вызов "hello".padStart(8) будет рястягивать строку "hello" на 8 символов. То есть изначально в строке "hello" 5 символов, значит, к ней будет добавлено 3 символа. При чем они будут добавлено в начале строки. По умолчанию добавляемые символы представляют пробелы. Аналогично вызов "hello".padEnd(8) растянет строку на 8 символов, но оставшие символы в виде пробелов будут добавлены в конец строки.

По умолчанию эти методы используют пробелы для заполнения, но в качестве второго параметра мы можем передать методам значение, которым надо дополнить строку:

let hello = "hello".padStart(17, "JavaScript, "); // "JavaScript, hello"

hello = "hello".padEnd(12, " Eugene"); // "hello Eugene"

Если добавляемое количество символов больше добавляемой строки, то добавляемая строка повторяется:

let hello = "123".padStart(6, "0"); // "000123"

hello = "123".padEnd(6, "0"); // "123000"

**Для чего используются методы .startsWith() и .endsWith()?**

Проверка начала и окончания строки

Метод startsWith() возвращает true, если строка начинается с определенной подстроки. А метод endsWith() возвращает true, если строка оканчивается на определенную подстроку.

const hello = "let me speak from my heart";

console.log(hello.startsWith("let")); // true

console.log(hello.startsWith("Let")); // false

console.log(hello.startsWith("lets")); // false

console.log(hello.endsWith("heart")); // true

console.log(hello.startsWith("bart")); // false

При этом играет роль регистр символов, и из примера выше мы видим, что "let" не эквивалентно "Let".

Дополнительный второй параметр позволяет указать индекс (для startsWith - индекс с начала, а для endsWith - индекс с конца строки), относительно которого будет производиться сравнение:

const hello = "let me speak from my heart";

console.log(hello.startsWith("me", 4)); // true, "me" - 4 индекс с начала строки

console.log(hello.startsWith("my", hello.length-8)); // true, "my" - 8 индекс с конца

**Как в JavaScript удалять пробельные символы в начале и в конце строки?**

Метод trim() удаляет пробельные символы с начала и конца строки

var orig = ' foo ';

console.log(orig.trim()); // 'foo'

**Расскажите об операторе Optional Chaining (?.)?**

Опциональная цепочка ?. останавливает вычисление и возвращает undefined, если значение перед ?. равно undefined или null.

**Далее в этой статье, для краткости, мы будем говорить, что что-то «существует», если оно не является null и не undefined.**

Другими словами, value?.prop:

* работает как value.prop, если значение value существует,
* в противном случае (когда value равно undefined/null) он возвращает undefined.

Вот безопасный способ получить доступ к user.address.street, используя ?.:

let user = {}; // пользователь без адреса

alert( user?.address?.street ); // undefined (без ошибки)

Код лаконичный и понятный, в нем вообще нет дублирования.

А вот пример с document.querySelector:

let html = document.querySelector('.elem')?.innerHTML; // будет undefined, если элемента нет

Считывание адреса с помощью user?.address работает, даже если объект user не существует:

let user = null;

alert( user?.address ); // undefined

alert( user?.address.street ); // undefined

Обратите внимание: синтаксис ?. делает необязательным значение перед ним, но не какое-либо последующее.

Так например, в записи user?.address.street.name ?. позволяет user безопасно быть null/undefined (и в этом случае возвращает undefined), но это так только для user. Доступ к последующим свойствам осуществляется обычным способом. Если мы хотим, чтобы некоторые из них были необязательными, тогда нам нужно будет заменить больше . на ?..

**Не злоупотребляйте опциональной цепочкой**

Нам следует использовать ?. только там, где нормально, что чего-то не существует.

К примеру, если, в соответствии с логикой нашего кода, объект user должен существовать, но address является необязательным, то нам следует писать user.address?.street, но не user?.address?.street.

В этом случае, если вдруг user окажется undefined, мы увидим программную ошибку по этому поводу и исправим её. В противном случае, если слишком часто использовать ?., ошибки могут замалчиваться там, где это неуместно, и их будет сложнее отлаживать.

**Переменная перед ?. должна быть объявлена**

Если переменной user вообще нет, то user?.anything приведёт к ошибке:

// ReferenceError: user is not defined

user?.address;

Переменная должна быть объявлена (к примеру, как let/const/var user или как параметр функции). Опциональная цепочка работает только с объявленными переменными.

**[Сокращённое вычисление](https://learn.javascript.ru/optional-chaining" \l "sokraschyonnoe-vychislenie)**

Как было сказано ранее, ?. немедленно останавливает вычисление, если левая часть не существует.

Так что если после ?. есть какие-то вызовы функций или операции, то они не произойдут.

Например:

let user = null;

let x = 0;

user?.sayHi(x++); // нет "user", поэтому выполнение не достигает вызова sayHi и x++

alert(x); // 0, значение не увеличилось

**[Другие варианты применения: ?.(), ?.[]](https://learn.javascript.ru/optional-chaining" \l "drugie-varianty-primeneniya)**

Опциональная цепочка ?. — это не оператор, а специальная синтаксическая конструкция, которая также работает с функциями и квадратными скобками.

Например, ?.() используется для вызова функции, которая может не существовать.

В приведённом ниже коде у некоторых наших пользователей есть метод admin, а у некоторых его нет:

let userAdmin = {

admin() {

alert("Я админ");

}

};

let userGuest = {};

userAdmin.admin?.(); // Я админ

userGuest.admin?.(); // ничего не произойдет (такого метода нет)

Здесь в обеих строках мы сначала используем точку (userAdmin.admin), чтобы получить свойство admin, потому что мы предполагаем, что объект user существует, так что читать из него безопасно.

Затем ?.() проверяет левую часть: если функция admin существует, то она запускается (это так для userAdmin). В противном случае (для userGuest) вычисление остановится без ошибок.

Синтаксис ?.[] также работает, если мы хотим использовать скобки [] для доступа к свойствам вместо точки .. Как и в предыдущих случаях, он позволяет безопасно считывать свойство из объекта, который может не существовать.

let key = "firstName";

let user1 = {

firstName: "John"

};

let user2 = null;

alert( user1?.[key] ); // John

alert( user2?.[key] ); // undefined

Также мы можем использовать ?. с delete:

delete user?.name; // удаляет user.name если пользователь существует

**Мы можем использовать ?. для безопасного чтения и удаления, но не для записи**

Опциональная цепочка ?. не имеет смысла в левой части присваивания.

Например:

let user = null;

user?.name = "John"; // Ошибка, не работает

// то же самое что написать undefined = "John"

**[Итого](https://learn.javascript.ru/optional-chaining" \l "itogo)**

Синтаксис опциональной цепочки ?. имеет три формы:

1. obj?.prop – возвращает obj.prop если obj существует, в противном случае undefined.
2. obj?.[prop] – возвращает obj[prop] если obj существует, в противном случае undefined.
3. obj.method?.() – вызывает obj.method(), если obj.method существует, в противном случае возвращает undefined.

Как мы видим, все они просты и понятны в использовании. ?. проверяет левую часть на null/undefined и позволяет продолжить вычисление, если это не так.

Цепочка ?. позволяет безопасно получать доступ к вложенным свойствам.

Тем не менее, мы должны использовать ?. осторожно, только там, где по логике кода допустимо, что левая часть не существует. Чтобы он не скрывал от нас ошибки программирования, если они возникнут.

<https://learn.javascript.ru/optional-chaining>

**Для чего используется метод .replaceAll()?**

Метод replaceAll() возвращает новую строку со всеми совпадениями pattern , который меняется на replacement . pattern может быть строкой или регулярным выражением, и replacement может быть строкой или функция возвращающая каждое совпадение.

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

**Что такое оператор логического присваивания?**

Оператор присваивания логического ИЛИ ( x ||= y ) присваивает значение переменной x только в случае, если её текущее значение ложноподобно.

**[Сокращённое вычисление](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Logical_OR_assignment" \l "сокращённое_вычисление)**

Оператор [логического ИЛИ](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Logical_OR) работает следующим образом:

x || y;

// возвращает x, если x истинноподобен

// возвращает y, если x не истинноподобен

Оператор логического ИЛИ OR производит вычисление по сокращённой схеме: второй операнд будет вычисляться только в случае, если при вычислении первого операнда не удалось получить значение.

Оператор присваивания логического ИЛИ также поддерживает сокращённое вычисление, т.е. выполнит операцию присваивания только при вычислении правого выражения. Проще говоря, x ||= y эквивалентно следующему коду:

x || (x = y);

В отличие от следующего примера, где операция присваивания произойдёт в любом случае:

x = x || y;

Обратите внимание, что математические и побитовые операторы присваивания работают по-разному.

**[Примеры](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Logical_OR_assignment" \l "примеры)**

**[Определение значения по умолчанию](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Logical_OR_assignment" \l "определение_значения_по_умолчанию)**

Если в элементе с идентификатором "lyrics" нет содержимого, то следующее выражение вставит в него надпись:

document.getElementById('lyrics').textContent ||= 'Нет слов.'

В данном примере сокращённое вычисление очень сильно выручает, поскольку элемент будет обновляться только при необходимости, что позволит избежать подобных эффектов вроде ненужного рендеринга, потери фокуса и т.д.

Примечание: обратите внимание на значение в левом операнде. Если это пустая строка ([ложноподобное](https://developer.mozilla.org/ru/docs/Glossary/Falsy) значение), то должно использовать оператор ||=, в противном случае — оператор ??= (если ожидается получение значения [null](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/null) или [undefined](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/undefined)).

**Как увеличить читаемость больших чисел?**

Как в джаве с \_

Числовые разделители, добавляя символы подчеркивания в алгебраических числах, делают их более удобочитаемыми. При анализе файлов эти символы подчеркивания автоматически удаляются. Следующий фрагмент кода — наглядный пример использования числовых разделителей.

// Десятичный целочисленный литерал с цифрами, сгруппированными по тысячам.  
let n1 = 1\_000\_000\_000;  
console.log(n1); // This will print: 1000000000  
  
// Десятичный литерал с цифрами, сгруппированными по тысячам.  
let n2 = 1\_000\_000\_000.150\_200  
console.log(n2); // This will print: 1000000000.1502  
  
// Шестнадцатеричный целочисленный литерал с цифрами, сгруппированными по байтам.  
let n3 = 0x95\_65\_98\_FA\_A9  
console.log(n3); // Будет выведено следующее: 641654651561  
  
// Литерал BigInt с цифрами, сгруппированными по тысячам.  
let n4 = 155\_326\_458\_156\_248\_168\_514n  
console.log(n4); // Будет выведено следующее: 155326458156248168514n

**Что такое приватные аксессоры?**

Методы и свойства класса по умолчанию являются публичными, но приватные методы и свойства могут быть созданы с помощью хэш-префикса #. Инкапсуляция приватности была введена в обновленном ECMAScript 2021. Доступ к этим приватным методам и свойствам возможен только изнутри класса. Следующий фрагмент кода поможет понять, как использовать приватные методы.

// Давайте создадим класс с именем User.  
class User {  
 constructor() {}  
  
 // Приватные методы можно создать, добавив ‘#’  
 // перед именем метода.  
 #generateAPIKey() {  
 return "d8cf946093107898cb64963ab34be6b7e22662179a8ea48ca5603f8216748767";  
 }  
  
 getAPIKey() {  
 // Доступ к приватным методам можно получить, используя ‘#’  
 // перед именем метода.  
 return this.#generateAPIKey();  
 }  
}  
  
const user = new User();  
const userAPIKey = user.getAPIKey();  
console.log(userAPIKey); // Будет выведено следующее: d8cf946093107898cb64963ab34be6b7e22662179a8ea48ca5603f8216748767

Приватные аксессоры — это приватные геттеры и сеттеры. Геттер позволяет получить значение свойства класса, а сеттер — присвоить значение свойству класса. Вы можете определить приватный геттер, используя хэш-префикс #.

get #newAccountPassword() {}

Аналогично, путем использования хэш-префикса #, можно определить приватный сеттер:

set #generateAccountPassword(newPassword) {}

Следующий фрагмент кода послужит примером использования приватных геттеров и сеттеров.

// Давайте создадим класс с именем Str.  
class Str {  
 // Приватные атрибуты могут быть созданы путем добавления ‘#’  
 // перед именем атрибута.  
 #uniqueStr;  
  
 constructor() {}  
  
 // Приватные сеттеры можно создавать, добавив ‘#’  
 // перед именем сеттера.  
 set #generateUniqueStringByCustomLength(length = 24) {  
 const characters = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";  
 let randomStr = "";  
  
 for (let i = 0; i < length; i++) {  
 const randomNum = Math.floor(Math.random() \* characters.length);  
 randomStr += characters[randomNum];  
 }  
  
 this.#uniqueStr = randomStr;  
 }  
  
 // Публичный сеттер  
 set setRandomString(length) {  
 this.#generateUniqueStringByCustomLength = length;  
 }  
  
 // Приватный геттер можно создать, добавив ‘#’  
 // перед именем геттера.  
 get #fetchUniqueString() {  
 return this.#uniqueStr;  
 }  
  
 // Публичный геттер  
 get getRandomString() {  
 return this.#fetchUniqueString;  
 }  
}  
  
const str = new Str();  
// Вызов публичного сеттера, который затем получит доступ к приватному сеттеру  
// внутри класса.  
str.setRandomString = 20;  
  
// Вызов публичного геттера, который затем получит доступ к приватному геттеру  
// внутри класса.  
const uniqueStr = str.getRandomString;  
console.log(uniqueStr); // Это будет выводить случайную строку каждый раз, когда вы выполняете геттер после сеттера.

<https://learn.javascript.ru/private-protected-properties-methods>

<https://metanit.com/web/javascript/4.16.php>

**Разница между ES6-классами и конструкторами функций?**

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

Классы — это относительно новый способ написания функций-конструкторов в JS. Это синтаксический сахар для функций-конструкторов. В основе классов лежат те же прототипы и прототипное наследование:

// ES5

**function** **Person**(firstName, lastName, age, address){

this.firstName = firstName

this.lastName = lastName

this.age = age

this.address = address

}

Person.self = **function**(){

**return** this

}

Person.prototype.toString = **function**(){

**return** '[object Person]'

}

Person.prototype.getFullName = **function**(){

**return** this.firstName + ' ' + this.lastName

}

// ES6

**class** **Person**{

**constructor**(firstName, lastName, age, address){

this.firstName = firstName

this.lastName = lastName

this.age = age

this.address = address

}

**static** **self**(){

**return** this

}

**toString**(){

**return** '[object Person]'

}

**getFullName**(){

**return** `${this.firstName} ${this.lastName}`

}

}

Переопределение методов и наследование от другого класса:

// ES5

Employee.prototype = Object.create(Person.prototype)

**function** **Employee**(firstName, lastName, age, address, jobTitle, yearStarted){

Person.call(this, firstName, lastName, age, address)

this.jobTitle = jobTitle

this.yearStarted = yearStarted

}

Employee.prototype.describe = **function**(){

**return** `I am ${this.getFullName()} and I have a position of #{this.jobTitle} and I started at ${this.yearStarted}}`

}

Employee.prototype.toString = **function**(){

**return** '[object Employee]'

}

// ES6

**class** **Employee** **extends** **Person**{ // наследуемся от Person

**constructor**(firstName, lastName, age, address, jobTitle, yearStarted){

super(firstName, lastName, age, address)

this.jobTitle = jobTitle

this.yearStarted = yearStarted

}

**describe**(){

**return** `I am ${this.getFullName()} and I have a position of #{this.jobTitle} and I started at ${this.yearStarted}}`

}

**toString**(){ // переопределяем метод toString класса Person

**return** '[object Employee]'

}

}

Как узнать об использовании прототипов?

**class** **Something**{ }

**function** **AnotherSomething**(){ }

**const** **as** = **new** AnotherSomething()

**const** s = **new** Something()

console.log(**typeof** Something) // function

console.log(**typeof** AnotherSomething) // function

console.log(**as**.toString()) // [object Object]

console.log(a.toString()) // [object Object]

console.log(**as**.toString === Object.prototype.toString)

console.log(a.toString === Object.prototype.toString)

// в обоих случаях получаем true

// Object.prototype находится на вершине цепочки прототипов

// Something и AnotherSomething наследуют от Object.prototype

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

<https://www.w3schools.com/js/js_object_constructors.asp>

Разница между объявлением функции (function declaration) и объявлением класса (class declaration) в том, что объявление функции совершает подъём (hoisting), в то время как объявление класса — нет. Поэтому вначале необходимо объявить ваш класс и только затем работать с ним, а код же вроде следующего сгенерирует исключение типа ReferenceError.

**Что такое оператор нулевого слияния (??)?**

Оператор нулевого слияния ( ?? )

— это логический оператор, возвращающий значение правого операнда, если значение левого операнда содержит null или undefined , в противном случае возвращается значение левого операнда.

ператор нулевого слияния представляет собой два вопросительных знака ??.

Так как он обрабатывает null и undefined одинаковым образом, то для этой статьи мы введём специальный термин. Для краткости будем говорить, что значение «определено», если оно не равняется ни null, ни undefined.

Результат выражения a ?? b будет следующим:

* если a определено, то a,
* если a не определено, то b.

Иначе говоря, оператор ?? возвращает первый аргумент, если он не null/undefined, иначе второй.

Оператор нулевого слияния не является чем-то принципиально новым. Это всего лишь удобный синтаксис, как из двух значений получить одно, которое «определено».

Вот как можно переписать выражение result = a ?? b, используя уже знакомые нам операторы:

result = (a !== null && a !== undefined) ? a : b;

Теперь должно быть абсолютно ясно, что делает ??. Давайте посмотрим, где это может быть полезно.

Как правило, оператор ?? нужен для того, чтобы задать значение по умолчанию для потенциально неопределённой переменной.

Например, здесь мы отобразим user, если её значение не null/undefined, в противном случае Аноним:

let user;

alert(user ?? "Аноним"); // Аноним (user не существует)

А вот пример, когда user присвоено значение:

let user = "Иван";

alert(user ?? "Аноним"); // Иван (user существует)

Кроме этого, можно записать последовательность из операторов ??, чтобы получить первое значение из списка, которое не является null/undefined.

Допустим, у нас есть данные пользователя в переменных firstName, lastName или nickName. Все они могут не существовать, если пользователь решил не вводить соответствующие значение.

Мы хотели бы отобразить имя пользователя, используя одну из этих переменных, или показать «Аноним», если все они null/undefined.

Для этого воспользуемся оператором ??:

let firstName = null;

let lastName = null;

let nickName = "Суперкодер";

// показывает первое значение, которое определено:

alert(firstName ?? lastName ?? nickName ?? "Аноним"); // Суперкодер

**[Сравнение с ||](https://learn.javascript.ru/nullish-coalescing-operator" \l "sravnenie-s)**

Оператор ИЛИ || можно использовать для того же, что и ??, как это было показано в [предыдущей главе](https://learn.javascript.ru/logical-operators" \l "or-finds-the-first-truthy-value).

Например, если в приведённом выше коде заменить ?? на ||, то будет тот же самый результат:

let firstName = null;

let lastName = null;

let nickName = "Суперкодер";

// показывает первое истинное значение:

alert(firstName || lastName || nickName || "Аноним"); // Суперкодер

Исторически сложилось так, что оператор ИЛИ || появился первым. Он существует с самого начала в JavaScript, поэтому разработчики долгое время использовали его для таких целей.

С другой стороны, сравнительно недавно в язык был добавлен оператор нулевого слияния ?? – как раз потому, что многие были недовольны оператором ||.

Важное различие между ними заключается в том, что:

* || возвращает первое *истинное* значение.
* ?? возвращает первое *определённое* значение.

Проще говоря, оператор || не различает false, 0, пустую строку "" и null/undefined. Для него они все одинаковы, т.е. являются ложными значениями. Если первым аргументом для оператора || будет любое из перечисленных значений, то в качестве результата мы получим второй аргумент.

Однако на практике часто требуется использовать значение по умолчанию только тогда, когда переменная является null/undefined. Ведь именно тогда значение действительно неизвестно/не определено.

Рассмотрим следующий пример:

let height = 0;

alert(height || 100); // 100

alert(height ?? 100); // 0

* height || 100 проверяет height на ложное значение, оно равно 0, да, ложное.
  + поэтому результатом || является второй аргумент, т.е. 100.
* height ?? 100 проверяет, что переменная height содержит null/undefined, а поскольку это не так,
  + то результатом является сама переменная height, т.е. 0.

На практике нулевая высота часто является вполне нормальным значением, которое не следует заменять значением по умолчанию. Таким образом, ?? здесь как раз работает так, как нужно.

**[Приоритет](https://learn.javascript.ru/nullish-coalescing-operator" \l "prioritet)**

Приоритет оператора ?? такой же, как и у ||. Они оба равны 3 в [таблице на](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence" \l "Table) MDN.

Это означает, что, как и ||, оператор нулевого слияния ?? вычисляется до = и ?, но после большинства других операций, таких как +, \*.

Так что, в выражениях такого вида понадобятся скобки:

let height = null;

let width = null;

// важно: используйте круглые скобки

let area = (height ?? 100) \* (width ?? 50);

alert(area); // 5000

Иначе, если опустить скобки, оператор \* выполнится первым, так как у него приоритет выше, чем у ??, и это приведёт к неправильным результатам.

// без скобок

let area = height ?? 100 \* width ?? 50;

// ...сработает вот так (совсем не как нам нужно):

let area = height ?? (100 \* width) ?? 50;

**[Использование ?? вместе с && или ||](https://learn.javascript.ru/nullish-coalescing-operator" \l "ispolzovanie-vmeste-s-ili)**

По соображениям безопасности JavaScript запрещает использование оператора ?? вместе с && и ||, если приоритет явно не указан при помощи круглых скобок.

Выполнение следующего кода приведёт к синтаксической ошибке:

let x = 1 && 2 ?? 3; // Синтаксическая ошибка

Это, безусловно, спорное ограничение было добавлено в спецификацию языка с целью избежать программные ошибки, когда люди начнут переходить с || на ??.

Используйте скобки, чтобы обойти это ограничение:

let x = (1 && 2) ?? 3; // Работает без ошибок

alert(x); // 2

**[Итого](https://learn.javascript.ru/nullish-coalescing-operator" \l "itogo)**

* Оператор нулевого слияния ?? — это быстрый способ выбрать первое «определённое» значение из списка.

Используется для присвоения переменным значений по умолчанию:

// будет height=100, если переменная height равна null или undefined

height = height ?? 100;

* Оператор ?? имеет очень низкий приоритет, лишь немного выше, чем у ? и =, поэтому при использовании его в выражении, скорее всего, потребуются скобки.
* Запрещено использовать вместе с || или && без явно указанного приоритета, то есть без скобок.

**В чём отличие оператора нулевого слияния (??) и оператора “ИЛИ” (||)?**

Важное различие между ними заключается в том, что:

* || возвращает первое *истинное* значение.
* ?? возвращает первое *определённое* значение.

Проще говоря, оператор || не различает false, 0, пустую строку "" и null/undefined. Для него они все одинаковы, т.е. являются ложными значениями. Если первым аргументом для оператора || будет любое из перечисленных значений, то в качестве результата мы получим второй аргумент.

Однако на практике часто требуется использовать значение по умолчанию только тогда, когда переменная является null/undefined. Ведь именно тогда значение действительно неизвестно/не определено.

**Назовите основные методы и свойства работы с коллекцией Map?**

[Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map) – это коллекция ключ/значение, как и Object. Но основное отличие в том, что Map позволяет использовать ключи любого типа.

Методы и свойства:

* [new Map()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/Map) – создаёт коллекцию.
* [map.set(key, value)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/set) – записывает по ключу key значение value.
* [map.get(key)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/get) – возвращает значение по ключу или undefined, если ключ key отсутствует.
* [map.has(key)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/has) – возвращает true, если ключ key присутствует в коллекции, иначе false.
* [map.delete(key)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/delete) – удаляет элемент (пару «ключ/значение») по ключу key.
* [map.clear()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/clear) – очищает коллекцию от всех элементов.
* [map.size](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/size) – возвращает текущее количество элементов.

<https://www.w3schools.com/js/js_maps.asp>

**Назовите основные методы и свойства работы с коллекцией Set?**

Объект [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set) – это особый вид коллекции: «множество» значений (без ключей), где каждое значение может появляться только один раз.

Его основные методы это:

* [new Set(iterable)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/Set) – создаёт Set, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый Set.
* [set.add(value)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/add) – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект set.
* [set.delete(value)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/delete) – удаляет значение, возвращает true, если value было в множестве на момент вызова, иначе false.
* [set.has(value)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/has) – возвращает true, если значение присутствует в множестве, иначе false.
* [set.clear()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/clear) – удаляет все имеющиеся значения.
* [set.size](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/size) – возвращает количество элементов в множестве.

Основная «изюминка» – это то, что при повторных вызовах set.add() с одним и тем же значением ничего не происходит, за счёт этого как раз и получается, что каждое значение появляется один раз.

**Как осуществить перебор элементов в коллекциях Map и Set?**

Для перебора коллекции Map есть 3 метода:

* [map.keys()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/keys) – возвращает итерируемый объект по ключам,
* [map.values()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/values) – возвращает итерируемый объект по значениям,
* [map.entries()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/entries) – возвращает итерируемый объект по парам вида [ключ, значение], этот вариант используется по умолчанию в for..of.

Например:

let recipeMap = new Map([

["огурец", 500],

["помидор", 350],

["лук", 50]

]);

// перебор по ключам (овощи)

for (let vegetable of recipeMap.keys()) {

alert(vegetable); // огурец, помидор, лук

}

// перебор по значениям (числа)

for (let amount of recipeMap.values()) {

alert(amount); // 500, 350, 50

}

// перебор по элементам в формате [ключ, значение]

for (let entry of recipeMap) { // то же самое, что и recipeMap.entries()

alert(entry); // огурец,500 (и так далее)

}

**Используется порядок вставки**

В отличие от обычных объектов Object, в Map перебор происходит в том же порядке, в каком происходило добавление элементов.

Кроме этого, Map имеет встроенный метод forEach, схожий со встроенным методом массивов Array:

// выполняем функцию для каждой пары (ключ, значение)

recipeMap.forEach((value, key, map) => {

alert(`${key}: ${value}`); // огурец: 500 и так далее

});

**[Object](https://learn.javascript.ru/map-set" \l "object-entries-map-iz-object).entries: Map из Object**

При создании Map мы можем указать массив (или другой итерируемый объект) с парами ключ-значение для инициализации, как здесь:

// массив пар [ключ, значение]

let map = new Map([

['1', 'str1'],

[1, 'num1'],

[true, 'bool1']

]);

alert( map.get('1') ); // str1

Если у нас уже есть обычный объект, и мы хотели бы создать Map из него, то поможет встроенный метод [Object.entries(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/entries), который получает объект и возвращает массив пар ключ-значение для него, как раз в этом формате.

Так что мы можем создать Map из обычного объекта следующим образом:

let obj = {

name: "John",

age: 30

};

let map = new Map(Object.entries(obj));

alert( map.get('name') ); // John

Здесь Object.entries возвращает массив пар ключ-значение: [ ["name","John"], ["age", 30] ]. Это именно то, что нужно для создания Map.

**[Object](https://learn.javascript.ru/map-set" \l "object-fromentries-object-iz-map).fromEntries: Object из Map**

Мы только что видели, как создать Map из обычного объекта при помощи Object.entries(obj).

Есть метод Object.fromEntries, который делает противоположное: получив массив пар вида [ключ, значение], он создаёт из них объект:

let prices = Object.fromEntries([

['banana', 1],

['orange', 2],

['meat', 4]

]);

// prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2

Мы можем использовать Object.fromEntries, чтобы получить обычный объект из Map.

К примеру, у нас данные в Map, но их нужно передать в сторонний код, который ожидает обычный объект.

Вот как это сделать:

let map = new Map();

map.set('banana', 1);

map.set('orange', 2);

map.set('meat', 4);

let obj = Object.fromEntries(map.entries()); // создаём обычный объект (\*)

// готово!

// obj = { banana: 1, orange: 2, meat: 4 }

alert(obj.orange); // 2

Вызов map.entries() возвращает итерируемый объект пар ключ/значение, как раз в нужном формате для Object.fromEntries.

Мы могли бы написать строку (\*) ещё короче:

let obj = Object.fromEntries(map); // убрать .entries()

Это то же самое, так как Object.fromEntries ожидает перебираемый объект в качестве аргумента, не обязательно массив. А перебор map как раз возвращает пары ключ/значение, так же, как и map.entries(). Так что в итоге у нас будет обычный объект с теми же ключами/значениями, что и в map.

**[Перебор объекта](https://learn.javascript.ru/map-set" \l "perebor-obekta-set) Set**

Мы можем перебрать содержимое объекта set как с помощью метода for..of, так и используя forEach:

let set = new Set(["апельсин", "яблоко", "банан"]);

for (let value of set) alert(value);

// то же самое с forEach:

set.forEach((value, valueAgain, set) => {

alert(value);

});

Заметим забавную вещь. Функция в forEach у Set имеет 3 аргумента: значение value, потом *снова то же самое значение* valueAgain, и только потом целевой объект. Это действительно так, значение появляется в списке аргументов дважды.

Это сделано для совместимости с объектом Map, в котором колбэк forEach имеет 3 аргумента. Выглядит немного странно, но в некоторых случаях может помочь легко заменить Map на Set и наоборот.

Set имеет те же встроенные методы, что и Map:

* [set.keys()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/keys) – возвращает перебираемый объект для значений,
* [set.values()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/values) – то же самое, что и set.keys(), присутствует для обратной совместимости с Map,
* [set.entries()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/entries) – возвращает перебираемый объект для пар вида [значение, значение], присутствует для обратной совместимости с Map.