

JavaScript

Глобальные объекты

Глобальные объекты

Глобальными называют переменные и функции, которые не находятся внутри какой-то функции. То есть, иными словами, если переменная или функция не находятся внутри конструкции `function`, то они «глобальные».

В JS все глобальные переменные и функции являются свойствами специального объекта, который называется «Глобальный объект» (Global object). В браузере этот объект доступен под именем `window`.

В браузере все глобальные объекты являются свойствами объекта `window`.



Date

Date

```
const date = new Date()
```

```
console.log(date) // покажет текущую дату и время
```

Параметры конструктора:

- **New Date(milliseconds)** – создаёт объект с временем, равным количеству миллисекунд, прошедших с 1 января 1970 года UTC+0
- **New Date(datestring)** – если аргумент один и это строка, то из неё прочитается дата.
- **New Date(year, month, date, hours, minutes, seconds, ms)** – создаёт объект с заданными компонентами в местном часовом поясе. Обязательны только первые два

Date. Методы

Получение компонентов даты:

- **getFullYear()** – получить год (4 цифры)
- **getMonth()** – получить месяц (от 0 до 11)
- **getDate()** – получить день месяца (от 1 до 31)
- **getHours()** – получить часы
- **getMinutes()** – получить минуты и тд.
- **getDay()** – получить день недели (от 0 до 6, где 0 – воскресенье)

Все эти методы возвращают значения в соответствии с местным часовым поясом.

Date. Методы

Установка компонентов даты:

- **setFullYear** – устанавливает год
- **setMonth** – устанавливает месяц
- **setDate** – устанавливает день месяца
- **setHours** – устанавливает часы
- **setMinutes** – устанавливает минуты
- **setSeconds** – устанавливает секунды

Date. Парсинг строки с датой

`Date.parse(str)` – считывает дату из строки

Формат должен быть вида: **YYYY-MM-DDTHH:mm:ss.sssZ**

- **YYYY-MM-DD** – год, месяц, день
- **T** – используется в качестве разделителя
- **HH:mm:ss.sss** – часы, минуты, секунды и миллисекунды
- **Z** – необязательная часть строки, которая обозначает часовой пояс в формате `+hh:mm`

Можно использовать и сокращённые варианты: **YYYY-MM-DD**, **YYYY-MM**, **YYYY**



Math

Math

Math – встроенный объект, хранящий в себе различные математические константы и функции. Math не умеет работать с числами типа BigInt

В отличие от Date его не нужно создавать, все методы и свойства являются статическими.

Math

Некоторые свойства объекта:

- **Math.E** – число Эйлера или непра (2.718)
- **Math.LN2** – натуральный логарифм из 2 (0,693)
- **Math.PI** – отношение длины окружности круга к его диаметру (3.14159)
- **Math.SQRT2** – квадратный корень из 2 (1.414)

Math

Некоторые методы объекта:

- **Math.abs(x)** – возвращает абсолютное значение числа
- **Math.cos(x)** – возвращает косинус числа
- **Math.sin(x)** – возвращает синус числа
- **Math.tan(x)** – возвращает тангенс числа
- **Math.floor(x)** – возвращает число, округлённое к меньшему целому
- **Math.log(x)** – возвращает натуральный логарифм числа
- **Math.pow(x,y)** – возвращает основание в степени экспоненты
- **Math.round(x)** – возвращает число, округлённое до ближайшего целого
- **Math.random()** – возвращает псевдослучайное число от 0 до 1

Location

Location

Location – объект, который содержит информацию о расположении текущей веб-страницы.

> location

```
< ▼ Location {ancestorOrigins: DOMStringList, href: "https://yandex.ru/", origin: "https://yandex.ru", protocol: "https:", host: "yandex.ru", ...} ⓘ  
  ▶ ancestorOrigins: DOMStringList {length: 0}  
  ▶ assign: f assign()  
    hash: ""  
    host: "yandex.ru"  
    hostname: "yandex.ru"  
    href: "https://yandex.ru/"  
    origin: "https://yandex.ru"  
    pathname: "/"  
    port: ""  
    protocol: "https:"  
  ▶ reload: f reload()  
  ▶ replace: f replace()  
    search: ""  
  ▶ toString: f toString()  
  ▶ valueOf: f valueOf()  
    Symbol(Symbol.toPrimitive): undefined  
  ▶ __proto__: Location
```

>

Location

Свойства:

- **href** – полная строка запроса к ресурсу. При изменении документ переходит на новую страницу
- **pathname** – содержит первый / после хоста с последующим текстом URL
- **origin** – содержит протокол, хост и порт текущего URL
- **protocol** – содержит протокол текущего URL
- **port** – номер порта
- **host** – хост, а именно имя хоста, :порт
- **hostname** – домен текущего URL
- **hash** - #идентификатор
- **search** - ? с параметрами URL

Location

Методы:

- **assign(url)** – загружает ресурс по URL, указанному в параметре
- **reload(forceReload)** – перезагружает ресурс по текущему URL (при добавлении true в параметры – будет выгружена с сервера, иначе с кеша)
- **replace(url)** – заменяет текущий ресурс на новый по URL, указанному в параметре. Отличие от **assign()** заключается в том, что **replace()** не сохранит текущую страницу в истории и пользователь не сможет нажать кнопку «назад»
- **toString()** – строка, содержащая URL целиком

DOM дерево

Document Object Model

Теги являются основой HTML-документа. Каждый тег является объектом, а вложенные теги являются детьми родительских элементов. Все объекты доступны при помощи JS и их можно использовать для изменения страницы.

Все теги являются узлами-объектами, корневым узлом является `<html>`.

Текст внутри элементов образует текстовые узлы. Текстовый узел содержит в себе только строку текста и у него не может быть потомков и он всегда находится на самом нижнем уровне.

Пробелы и переводы строки – это тоже символы, как буквы и цифры. Они тоже образуют текстовые узлы и становятся частью дерева **DOM**.

Document Object Model

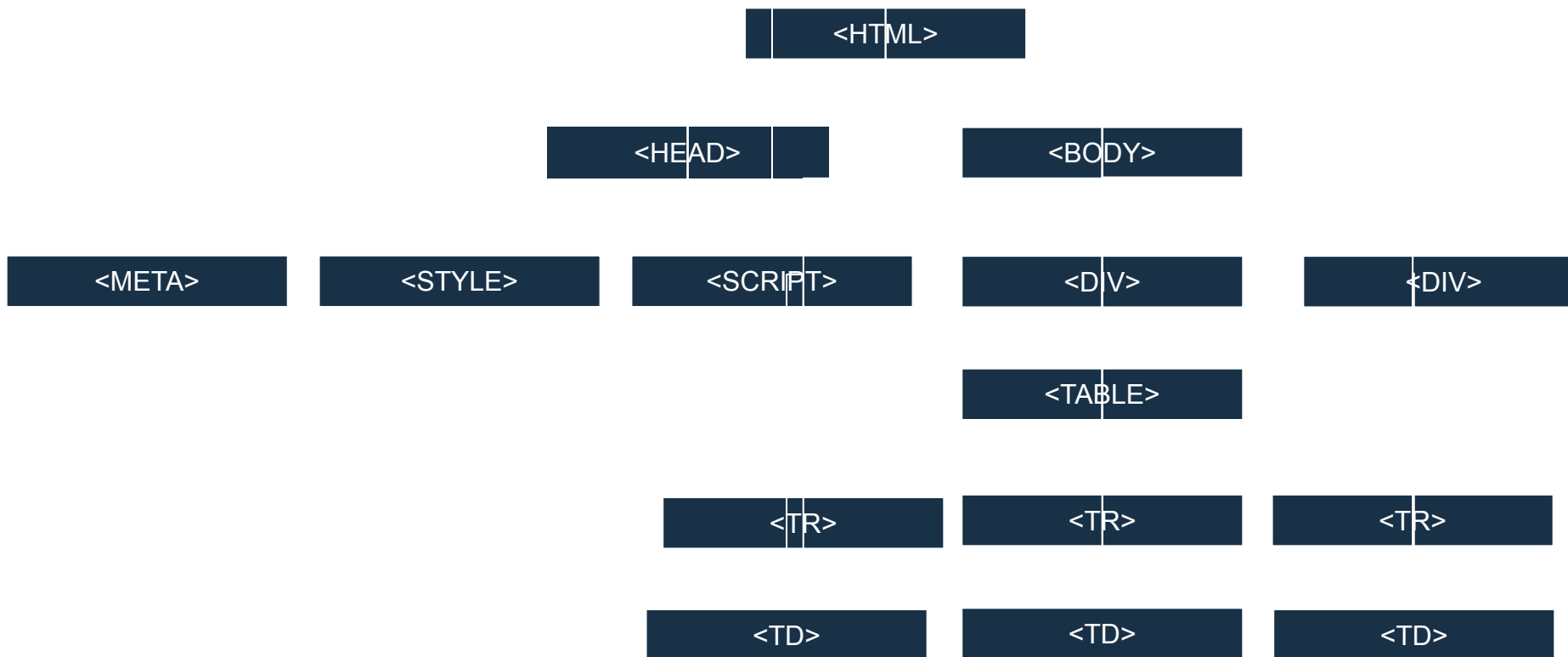
Исключения:

- Пробелы и перевод строки перед тегом **<head>** игнорируется
- При записи чего либо после **</body>**, браузер автоматически перемещает эту запись в конец body. Это происходит из-за того, что спецификация HTML требует, чтобы всё содержимое было внутри **<body></body>**.

В остальных случаях всё предельно просто. Если в документе есть пробелы, они становятся текстовыми узлами дерева **DOM** и если мы их удалим, то из **DOM** они тоже пропадут.

Document Object Model

В графическом виде DOM можно представить в виде дерева:



Document Object Model

Document – является входной точкой в **DOM**.

Если вывести в консоль **document**, то мы увидим следующую структуру:

```
> document
< ▼#document
  <!DOCTYPE html>
  <html class="i-ua_js_yes i-ua_css_standart i-ua_browser_chrome i-ua_browser-engine_webkit i-ua_browser_desktop font_loaded document_infinity-zen-loading_yes i-ua_platform_windows js i-ua_placeholder_yes i-ua_inlinesvg_yes m-svg i-ua_animation_yes i-ua_user-font-size_normal i-ua_user-font-size_16px i-ua_retina_yes utilityfocus m-stat fonts-loaded" lang="ru">
    ▶ <head xmlns:og="http://ogp.me/ns#">...</head>
    ▼ <body class="not-logged b-page_infinity-zen_yes i-ua i-bem b-page b-page_search-bottom_yes b-page_hide-messenger-button_yes i-ua_platform_other i-ua_js_initied" data-bem="{i-ua:}">
      ▶ <svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" style="position: absolute; width: 0; height: 0" id="__SVG_SPRITE_NODE__">...</svg>
      ▶ <div class="body_wrapper">...</div>
      <div class="style-checker"></div>
      ▶ <div class="mini-suggest_popup mini-suggest_popup_svg_yes mini-suggest_popup_theme_flat" style="top: 344.8px; width: 827px; left: 302.6px;">...</div>
      ▶ <div aria-hidden="true" style="position: absolute; left: -10px; top: 0; width: 1px; height: 1px; overflow: hidden;">...</div>
    </body>
  </html>
>
```

Document Object Model

Пример взаимодействия с **DOM** через код:

```
function newDiv() {  
    var element = document.createElement("div");  
    var body = document.getElementsByTagName("body")[0];  
    element.setAttribute("class", "textBox");  
    element.innerHTML = "Hello, I'm DIV tag!";  
    body.appendChild(element);  
}
```

Основные итерфейсы. Работа с document

- **document.querySelector("#foo")** – возвращает первый элемент документа, который соответствует указанному селектору. Если совпадений нет – null
- **document.querySelectorAll("p, .class")** – возвращает все элементы, соответствующие селектору
- **document.getElementById(id)** – возвращает ссылку на элемент по его идентификатору.
- **document.getElementsByTagName(tagName)** – возвращает HTMLCollection элементов с указанным именем тега.
- **document.getElementsByClassName(className)** – возвращает HTMLCollection элементов, соответствующих всем из указанных имён классов.
- **document.createElement(tagName, [options])** – создаёт элемент.

Основные интерфейсы. QuerySelector

Зачем нужны остальные, когда есть querySelector?

```
const elements1 = document.querySelectorAll('div')
const elements2 = document.getElementsByTagName('div')
const newElement = document.createElement('div')
```

```
document.body.appendChild(newElement)
elements1.length === elements2.length
```


Основные итерфейсы. QuerySelector

Результат вывода `querySelector()` не обновляется. Т.е. когда мы добавим новый элемент, старый список не изменится.

`querySelector()` обладает несколькими минусами:

- Так как результат собирается в список, то к нему нельзя просто применить какие-то методы.
- Список не является массивом и к нему нельзя применить методы массивов. Так что для удобной работы лучше преобразовывать в массив

У каждого элемента есть некоторые свойства, ссылающиеся на семью

- `children`
- `firstElementChild` / `lastElementChild`
- `childNodes` / `parentElement`

Основные итерфейсы. Элементы

- **element.innerHTML** – устанавливает или получает HTML разметку дочерних элементов
- **element.style.color = "blue"** – устанавливает или получает инлайн стили
- **element.setAttribute(attrName, value)** – добавляет новый атрибут или изменяет значение существующего
- **element.getAttribute(attrName)** – возвращает значение указанного атрибута
- **element.addEventListener(type, listener)** – регистрирует обработчик события
- **element.classList** – позволяет добавить или удалить класс (.add / .remove)

Основные итерфейсы. Изменение DOM

```
const newElement = document.createElement('div')
```

- `element.appendChild(newElement)` – добавление `newElement` как последнего дочернего у `element`
- `element.insertBefore(newElement, oldElement)` – вставка `newElement` как дочернего у `element` перед `oldElement`
- `element.cloneNode(true)` – создание клона (при `true` – копируются и дети)
- `parentElement.removeChild(newElement)` – удаление элемента

Обработчики событий

```
div.onclick = () => 1
```

Таких обработчиков лучше избегать, т.к. мы заменяем свойство элемента. В таком случае мы не сможем добавить другие обработчики используя ещё одну функцию, ссылаясь на старую.

Для добавления обработчиков лучше использовать:

```
div.addEventListener('click', (event) => console.log())
```

Удаление обработчика:

```
div.removeEventListener('click', (event) => console.log())
```

Предотвращение действий по умолчанию

Для предотвращения действий по умолчанию используется метод **.preventDefault()**, который блокирует стандартные действия. Например он заблокирует отправку формы, если авторизация на клиентской стороне не была успешной.

.stopPropagation() поможет, если есть определённый обработчик события, закреплённый за дочерним элементом и второй обработчик того же события, закреплённый за родителем. Другими словами – предотвращает распространение события.

Области ВИДИМОСТИ

Область видимости

Область видимости является очень важной концепцией, определяющей доступность переменных.

```
const message = "message"  
console.log(message) // message
```

```
if (true) { const message = "message" }  
console.log(message) // ReferenceError: message is not defined
```

Так произошло, т.к. блок `if` в нашем случае создал область видимости для переменной и эта переменная доступна только внутри этой области. Таким образом, доступность переменных ограничена областью видимости, в которой они определены

Что такое замыкание?

Замыкание это функция у которой есть доступ к своей внешней функции по области видимости, даже после того, как внешняя функция завершилась. Это говорит нам о том, что замыкание запоминает и получает доступ к переменным и аргументам своей внешней функции даже после того как она прекратит своё выполнение.

Лексическая область видимости состоит из внешних областей и это статическая область в JS, имеющая прямое отношение к доступу к переменным, функциям и объектам, основываясь на их расположении.

Лексическое окружение

Объект лексического окружения состоит из двух частей: объекта, в котором хранятся все локальные переменные и ссылка на внешнее лексическое окружение.

```
const name = "Name"  
function hello(message) {  
  console.log(name, message)  
}
```

В данном примере лексическое окружение у функции будет содержать переменную `name`. Внешнее лексическое окружение – это глобальное лексическое окружение. В нём находится наша переменная `name` и сама функция.

Лексическое окружение

Каждый раз, когда код захочет получить доступ к переменной – сначала будет происходить поиск по внутреннему лексическому окружению, затем по внешнему, затем ещё во внешнем и т.д.

При каждом новом вызове функции будет создаваться новое лексическое окружение!

Все функции при создании получают скрытое свойство `[[Environment]]`, которое ссылается на лексическое окружение места, где они были созданы.

Блочная область видимости

```
if (true) {  
  const message = "message"  
  console.log(message) // message  
}  
console.log(message) // ReferenceError: message is not defined
```

В конструкциях if, for, while создаётся блочная область видимости.

var не имеет блочной области видимости!

Область видимости функции

Функции создают свою область видимости для всех переменных.

```
function sayMyName() {  
  var name = "Name"  
  console.log(name) // Name  
}
```

```
console.log(name) // Reference Error
```

Вложенные области видимости

Области видимости могут быть вложены друг в друга

```
function sayMyName() {  
  const name = "Name"  
  if (true) {  
    const message = "message"  
    console.log(name, message) // Name message  
  }  
  console.log(message) // Reference Error  
}
```

Область видимости, находящаяся внутри другой называется внутренней областью видимости.

Глобальная область видимости

Глобальная область видимости это самая внешняя область. Она доступна любой внутренней области.

Переменные, которые были созданы в глобальной области видимости являются глобальными переменными. Таковыми являются, например **window** или **document**

Async\Defer

Асинхронные скрипты: defer/async

Когда браузер загружает HTML и доходит до скрипта, он не может продолжать строить DOM. Он должен сначала выполнить скрипт. Аналогичная ситуация происходит и с внешними скриптами: Браузер должен подождать, пока загрузится скрипт, выполнить его, и только потом обработать остальную страницу.

Такое поведение ведёт к двум большим и важным проблемам:

1. Скрипты не видят DOM-элементы ниже себя, поэтому к ним нельзя обратиться.
2. Если вверху страницы объёмный скрипт, то он блокирует страницу. Пользователи не видят содержимое страницы до тех пор, пока он не загрузится и не запустится.

Асинхронные скрипты: **defer/async**

Есть несколько вариантов как избежать такой ситуации. Самый простой – разместить скрипт внизу страницы. Таким образом он сможет видеть все элементы над ним и не будет препятствовать отображению страницы.

Такой вариант плох тем, что вёрстка может быть очень длинной и в таком случае будет заметна задержка, особенно при плохом интернет-соединении.

Следующий способ - использование атрибутов **async** и **defer** тега `<script>`

defer

Данный атрибут сообщает браузеру, что он должен продолжать обрабатывать страницу и загружать скрипт в фоновом режиме, а затем запустить скрипт, когда он загрузится.

```
<div> блок </div>  
  <script defer src=""></script>  
<div> блок2 </div>
```

Плюсы использования:

- не будут блокировать страницу
- Всегда выполняются, когда DOM готов, но до события **DOMContentLoaded**

DOMContentLoaded – ждёт отложенный скрипт

defer

Отложенные скрипты сохраняют последовательность выполнения. Именно поэтому если второй скрипт меньше первого и загрузится быстрее, то он будет ждать первый.

Если в теге `<script>` отсутствует **src**, то атрибут **defer** будет игнорироваться.

async

Данный атрибут говорит о том, что скрипт будет независимым, а именно:

- Страница не ждёт асинхронных скриптов, содержимое обрабатывается и отображается
- Событие **DOMContentLoaded** и асинхронные скрипты не ждут друг друга (может произойти как до, так и после асинхронного скрипта)
- Остальные скрипты не ждут **async** и скрипты с **async** тоже никого не ждут

```
<div> блок </div>
```

```
<script async src=""></script>
```

```
<script async src=""></script>
```

```
<div> блок2 </div>
```

Содержимое такой страницы отобразится сразу

async

```
<script async src = ""></script> //12MB
```

```
<script async src = ""></script> //1MB – выполнится первым, т.к. скачается быстрее
```

Такие скрипты чаще всего используют для подгрузки сторонних модулей, таких как счётчиков рекламы и тп.

Есть возможность и динамической подгрузки скриптов. Такие скрипты по умолчанию ведут себя как async.

```
const script = document.createElement('script');  
script.src = "source"  
document.body.append(script)
```

Q&A

Thank You



