

ВОЗМОЖНОСТИ JAVASCRIPT В БРАУЗЕРЕ



Владимир
Языков



ВЛАДИМИР ЯЗЫКОВ

Основатель UsefulWeb



ЧТО ВАС ЖДЁТ НА КУРСЕ

1. Научимся работать с web-интерфейсами;
2. Создадим несколько web-игр;
3. 24 обязательные задачи + необязательные повышенной сложности (для скучающих);
4. Дополнительные материалы даже в домашних заданиях;
5. Познаем внутренности современных фреймворков и библиотек;
6. Научимся менять содержимое страницы до неузнаваемости.



ПЛАН ЗАНЯТИЯ

1. [Знакомство с основными составляющими браузера](#)
2. [Как подключить JavaScript на страницу](#)
3. [Как браузер выводит страницу на экран](#)
4. [Объектная модель документа](#)
5. [Синхронное и асинхронное выполнение JavaScript](#)
6. [Работа с атрибутами html-элементов](#)
7. [Вызов функций после действия пользователя на странице](#)
8. [Немного про объектную модель браузера](#)

ЗНАКОМСТВО С ОСНОВНЫМИ СОСТАВЛЯЮЩИМИ БРАУЗЕРА

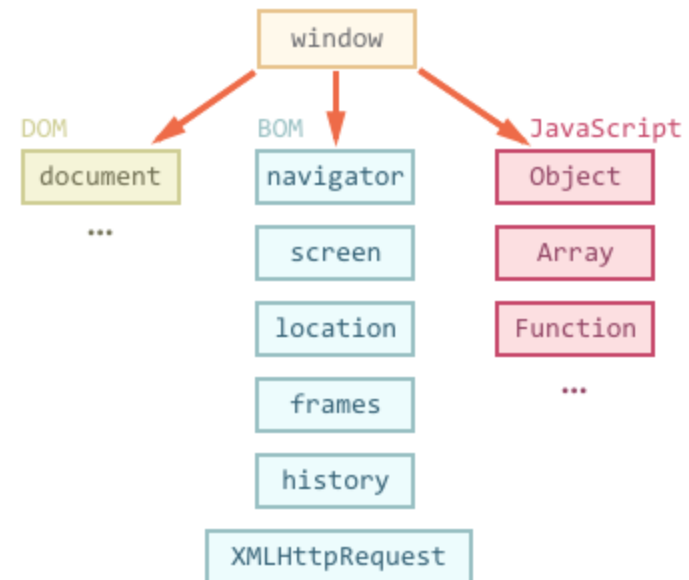
БРАУЗЕРЫ

Браузер — это программное обеспечение для отображения веб-страниц. Браузеры очень широко используются в различных устройствах: не только в привычных компьютерах и смартфонах, но и в телевизорах, игровых автоматах и даже терминалах оплаты.

Каждый браузер имеет объект `window` и три важных элемента в его структуре:

- **Объектную модель документа (DOM)** — при помощи которой мы можем получить доступ к содержимому страницы;
- **Объектную модель браузера (BOM)** — которая содержит различные функции для работы с браузером, но не с документом;


— **JavaScript** — который помимо своих основных функций имеет доступ к трём остальным элементам: `window`, DOM и BOM



НЕМНОГО О JAVASCRIPT В БРАУЗЕРЕ

JavaScript — не просто язык программирования, это реализация спецификации ECMA-262, которая описывает что, как и зачем должно работать. Это означает, что по мере того, как у спецификации появляются новые черновики или опубликованные редакции, разработчики браузеров и фреймворков вроде Node.js должны последовательно внедрять новый функционал. Для этого вносятся изменения в движки, которые эти браузеры и фреймворки используют для интерпретирования и выполнения кода JavaScript.

Есть множество редакций этой спецификации, но хотя сейчас актуальной считается 9 редакция (ES2018), некоторые браузеры все еще с горем пополам поддерживают редакцию 5 (ES5), которая была опубликована еще в 2009 году.



КАК ПОДКЛЮЧИТЬ JAVASCRIPT НА СТРАНИЦУ

ПОДКЛЮЧАЕМ JAVASCRIPT

Давайте посмотрим на очень простую HTML-страницу:

```
1 <html>
2   <head>
3     <title>Заголовок</title>
4   </head>
5   <body>
6     Привет, Мир!
7   </body>
8 </html>
```

В ней нет ничего, связанного с **JavaScript**, поэтому браузер просто отобразит строчку **Привет, Мир!** и на этом все.

Чтобы браузер начал исполнять какой-то **JavaScript** код, его необходимо поместить внутрь нашей страницы.

ПОДКЛЮЧИТЬ JAVASCRIPT НА СТРАНИЦУ МОЖНО ДВУМЯ СПОСОБАМИ

Первый способ — поместить код JavaScript внутри специального тега – `<script>` – и добавить этот тег `<script>` на страницу. Например, вот так:

```
1 <html>
2   <head>
3     <title>Заголовок</title>
4   </head>
5   <body>
6     Привет
7     <script>
8       console.log("Мир!");
9     </script>
10  </body>
11 </html>
```

Что будет делать браузер:

1. Отобразит содержимое HTML документа до тега `<script>`;
2. Исполнит содержимое тега `<script>` как JavaScript код;
3. Продолжит отображать содержимое, пока не встретит следующий тег `<script>` или пока не дойдет до конца.

Второй способ подключения JavaScript — написать скрипт в отдельном файле, например, `index.js` и потом подключить его к **html** странице.

Для подключения внешних скриптов необходимо использовать атрибут `src` тега `<script>`. Значением этого атрибута является путь до файла со скриптом. А пути бывают разных видов:

```
<script src="index.js"></script>
```

- Этот скрипт мы загрузили с использованием **относительного пути**, то есть `index.js` должен быть расположен в той же директории, что и загруженная **html** страница.

```
<script src="/scripts/library.js"></script>
```

- Здесь показан **абсолютный путь**. Он начинается с `/` и отсчитывается от корня сайта.

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.1.1/jquery.min.js"></script>
```

А это **полный URL** до какого-то скрипта, который, как правило, находится на другом сайте. Он начинается с `http://` или `https://`, далее идет доменное имя, например `ajax.googleapis.com`, а затем уже абсолютный путь до файла.

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/3.1.1/jquery.min.js"></script>
```

С помощью этого варианта можно загрузить внешний скрипт с другого сайта, но без точного указания протокола. Скрипт будет загружен по `http`, если текущая страница открыта с помощью `http`. Если же текущий протокол `https`, то и загрузка внешнего скрипта пойдет по `https`.

Существует один **неверный** вариант использования тега `<script>`, о котором необходимо упомянуть. Если используется одновременно атрибут `src` и содержимое тега, то содержимое будет проигнорировано.

```
<script src="index.js">  
  console.log("test"); // никогда не будет выведено  
</script>
```

КАК БРАУЗЕР ВЫВОДИТ СТРАНИЦУ НА ЭКРАН



КАК БРАУЗЕР ВЫВОДИТ СТРАНИЦУ НА ЭКРАН

От html-кода до полноценной страницы в браузере проходит несколько этапов:

- Построение объектной модели документа (DOM);
- Построение объектной модели CSS (CSSOM);
- Построение модели визуализации;
- Отрисовка макета страницы.



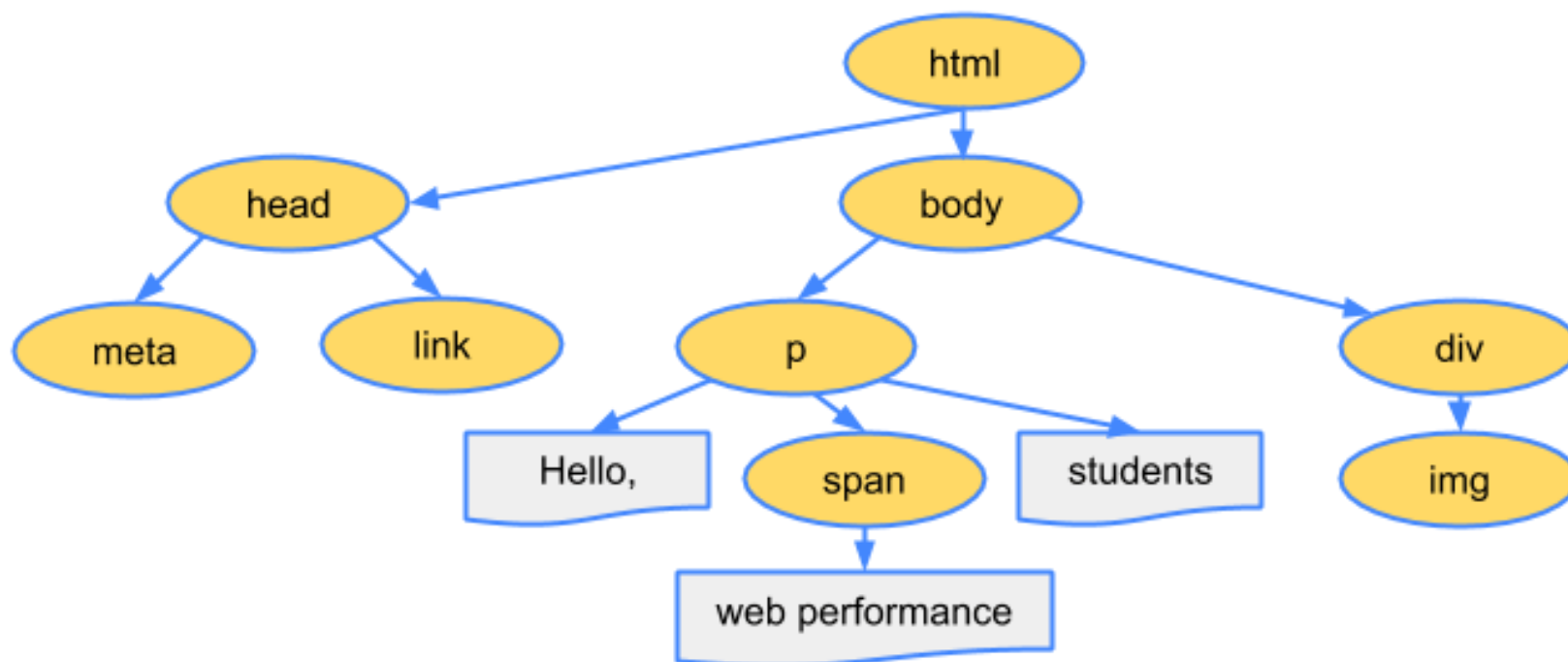
ОБЪЕКТНАЯ МОДЕЛЬ ДОКУМЕНТА

ПОСТРОЕНИЕ DOM

В первую очередь, получив html-файл, браузер строит объектную модель документа. Для этого он считывает html файл и проходит по всей его иерархии, создавая древовидную структуру.

В качестве примера давайте рассмотрим следующий HTML-код и его представление в виде DOM:

```
1 <html>
2   <head>
3     <meta charset="UTF-8">
4     <link rel="stylesheet" href="styles.css">
5   </head>
6   <body>
7     <p>
8       Hello <span>web perfomance</span> students
9     </p>
10    <div>
11      
12    </div>
13  </body>
14 </html>
```

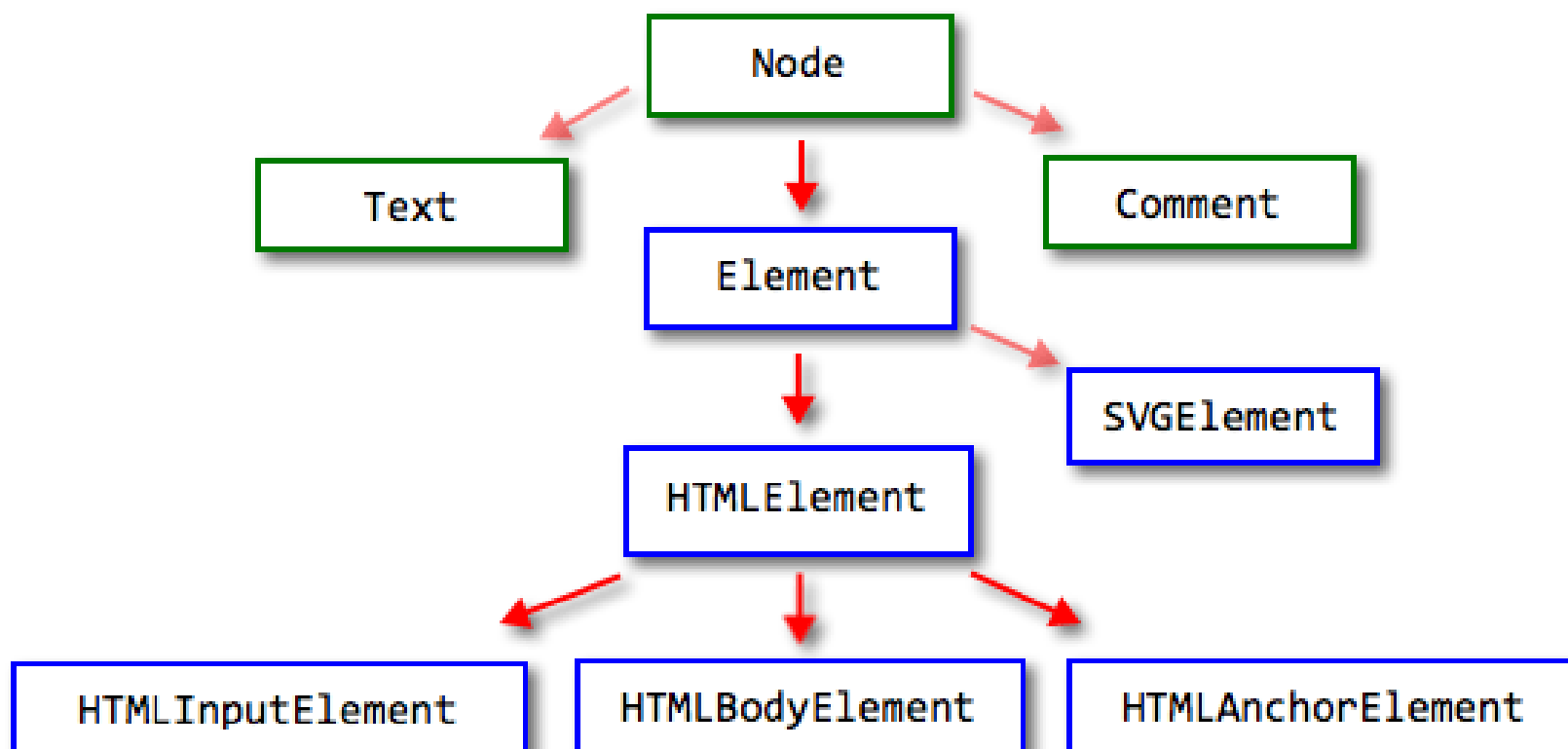


Как вы видите, иерархия DOM полностью повторяет иерархию html. Но состоит она из узлов и соединений, которые указывают на потомков конкретного узла. Узлами в данном случае выступают теги (желтые блоки) и текстовое содержимое этих тегов (голубые блоки). Или, по-другому, это *узлы-элементы* и *текстовые узлы*.

Мы из JavaScript можем влиять на DOM, но за изменением DOM последует также изменение модели визуализации и макета страницы.

НЕМНОГО ПРО УЗЛЫ

Узлы DOM имеют множество типов и классов, у которых также есть своя иерархия. Не будем подробно останавливаться на всех типах, поэтому посмотрим на основные:



Узел (`Node`) делится на три основных типа: `Text` , `Element` и `Comment` .

С первым мы уже знакомы — обычный **текстовый узел**. Также мы с вами знакомы с узлом `HTMLElement` — это основной тип всех `html-тегов` в `DOM` . Но он является лишь подтипом узла `Element` , у которого помимо него есть еще один — `SVGElement` . Как вы уже догадались — это базовый тип всех `svg-тегов` .

Сам же `HTMLElement` имеет множество подтипов, каждый из которых имеет свой и часто различный набор стандартных свойств и методов. Но также у всех них много общего, ведь все они являются потомками основного типа `HTMLElement` . Если в дальнейшем мы будем упоминать `Element` , то речь зачастую будет идти именно о `HTMLElement` .

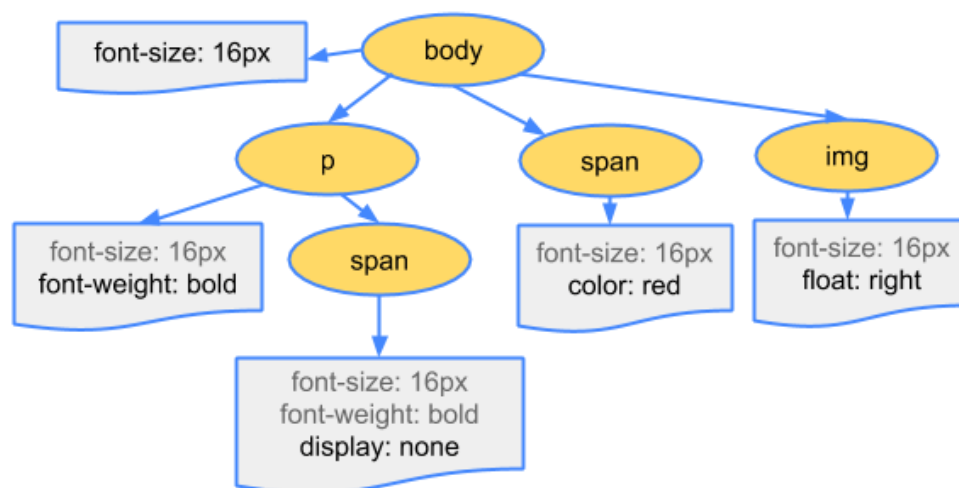
И третий, основной тип узла (`Comment`) — обычный HTML комментарий.

ПОСТРОЕНИЕ ОБЪЕКТНОЙ МОДЕЛИ CSS

Во время формирования `DOM` браузер обнаружил в документе ссылку на таблицу стилей (`style.css`). Поскольку стили являются неотъемлемой частью страницы, браузер запрашивает у сервера данные из этого файла, чтобы построить объектную модель CSS (таблицы стилей).

В ответ от сервера браузер получит следующий код и сразу создаст по нему объектную модель CSS (CSSOM):

```
1  body { font-size: 16px }
2  p { font-weight: bold }
3  span { color: red }
4  p span { display: none }
5  img { float: right }
```



Жёлтые узлы — теги, которые могут быть обнаружены в HTML, а голубые узлы — сами стили, которые будут к этим тегам применяться. Каждый элемент в цепочке сначала наследует стили своего родителя, а потом применяет свои собственные стили.

Обратите внимание, что данная схема из примера отображает только загруженные стили без стандартных стилей браузера.

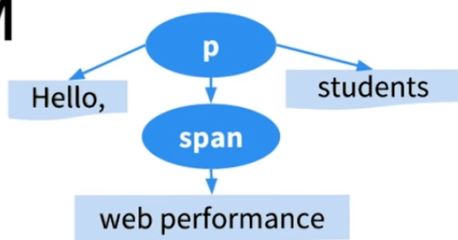
Мы из JavaScript можем также влиять и на CSSOM, но за изменением CSSOM точно также, как и за изменением DOM, последует обновление модели визуализации и макета страницы.

МОДЕЛЬ ВИЗУАЛИЗАЦИИ

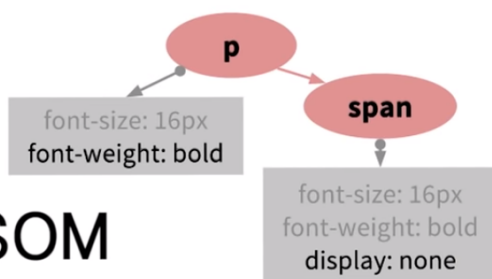
После построения DOM и CSSOM браузер приступает к объединению этих двух моделей, чтобы создать общую структуру, по которой можно будет начать отрисовывать сам макет страницы. Эта объединенная модель называется **моделью визуализации**.

Давайте взглянем на отдельный узел абзаца в наших моделях DOM и CSSOM, чтобы разобраться что к чему:

DOM

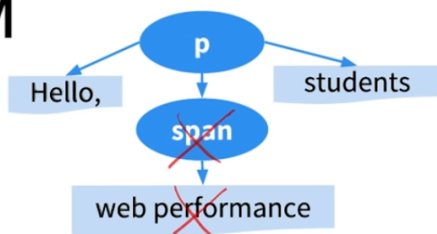


CSSOM

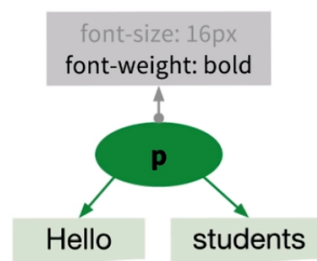


Объединив эти модели, мы получим следующую модель визуализации:

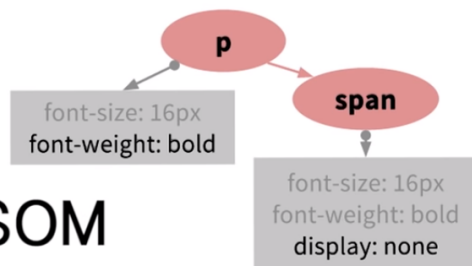
DOM



Модель визуализации

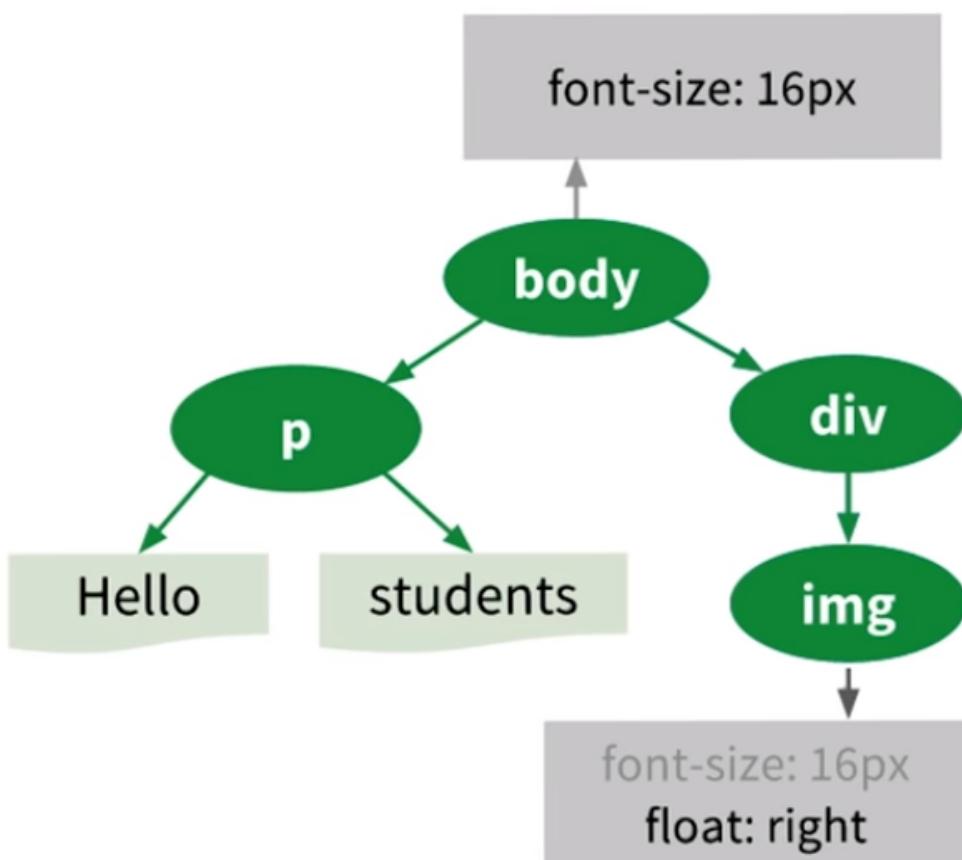


CSSOM



Не все элементы будут включены в модель визуализации. Браузеры не включают в модель информационные элементы, не предназначенные для пользователя (вроде `head` с его содержимым), а также элементы, которые были отключены при помощи стилей `display:none`.

Следующим образом будет выглядеть модель визуализации для всего нашего html-документа:



МАКЕТ СТРАНИЦЫ

Дальше происходит формирование страницы из модели визуализации, и единственное, о чем здесь можно упомянуть — откуда браузер будет рассчитывать ширину для `body`, если мы установим её, например, в `50%`.

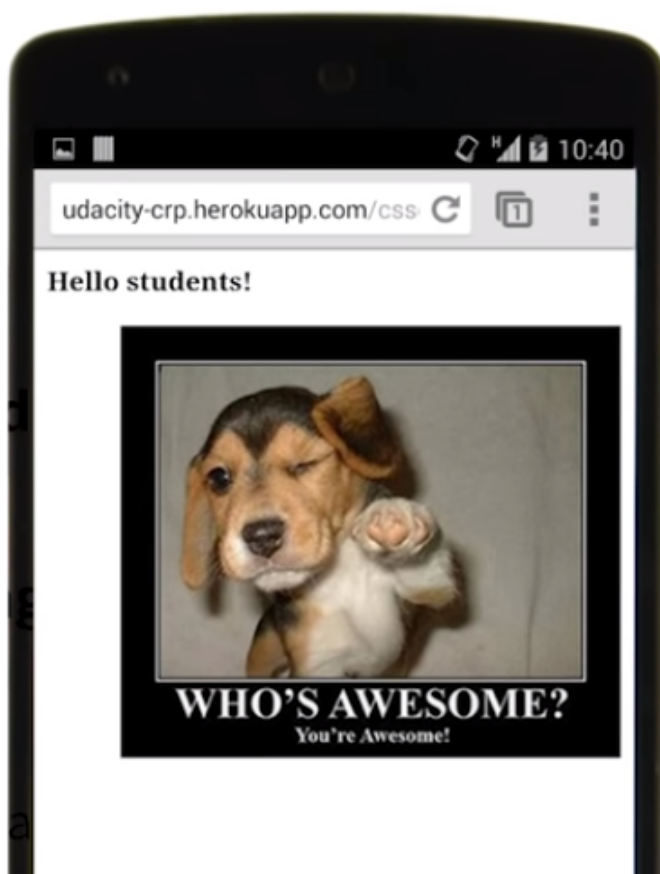
Если все потомки тега `body` берут за 100% ширину от своего родителя, то тег `body` за 100% берет ширину от `viewport`. Обычно она устанавливается через тег:

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

И ширина `viewport` в данном случае будет равна ширине устройства в пикселях. Но если этот тег не установлен, то браузер берет стандартное значение, которое обычно составляет `980px`.

ВЫВОД СТРАНИЦЫ НА ЭКРАН

И после всех вычислений браузер отрисовывает нашу страницу и показывает её содержимое:





ВОПРОСЫ

- В каком порядке браузер производит расчет следующих элементов: CSSOM, Модель визуализации, Макет страницы, DOM?

РЕЗЮМИРУЕМ

Чтобы вывести страницу на экран, браузер выполняет следующее:

- Строит объектную модель документа (DOM) на основе html кода страницы;
- Строит объектную модель CSS (CSSOM) на основе стилей, подключенных к странице;
- Строит модель визуализации, объединяя DOM и CSSOM, и выкидывая всё лишнее;
- Отрисовывает макет страницы, устанавливая ширину `viewport` в качестве 100% ширины для `body`.

СИНХРОННОЕ И АСИНХРОННОЕ ВЫПОЛНЕНИЕ JAVASCRIPT

СИНХРОННОЕ И АСИНХРОННОЕ ВЫПОЛНЕНИЕ JAVASCRIPT

Мы уже знаем, как подключается JavaScript на страницу, но нужно остановиться на таком нюансе, как полная остановка дальнейшей обработки DOM в браузере до тех пор, пока не выполнится код JavaScript.

Если браузер видит тег `<script>`, то он по стандарту обязан:

- Загрузить файл скрипта (если есть атрибут `src`);
- Выполнить его;
- Показать оставшуюся часть страницы.


```
1 <body>
2   <p>Hello</p>
3   <script src="script.js">
4   <!--
5       Все, что идет далее, будет выведено на страницу
6       только после выполнения script.js
7   -->
8   <p>students!</p>
9 </body>
```

Такое поведение называют «синхронным». Как правило, оно вполне нормально, но есть важное следствие.

Если скрипт – синхронный, то, пока браузер не выполнит его, он не покажет часть страницы под ним.

Решить эту проблему помогут атрибуты `async` или `defer`. Оба атрибута никак не влияют на встроенные в HTML скрипты, т.е. на те, у которых нет атрибута `src`.

АТРИБУТ `defer`

Атрибут `defer` поддерживается всеми браузерами, включая самые старые IE. Скрипт выполняется асинхронно, но браузер гарантирует, что порядок скриптов с `defer` будет сохранён — они будут выполняться последовательно в том порядке, в котором расположены в документе. Первым выполнится код из файла `first.js`, а `second.js` вторым:

```
<script src="./first.js" defer></script>  
<script src="./second.js" defer></script>
```

Скрипт `second.js`, даже если загрузился раньше, будет ожидать выполнения синхронной части кода из `first.js`.

Поэтому атрибут `defer` используют в тех случаях, когда код одного скрипта использует ресурсы другого скрипта. Например, если мы подключаем библиотеку и скрипт, который её использует, и хотим их подключить асинхронно, то должны использовать `defer`, и файл библиотеки подключить первым:

```
<script src="./lib.js" defer></script>  
<script src="./client.js" defer></script>
```

Также важной особенностью скрипта, подключенного с атрибутом `defer`, является его исполнение после того, как браузер полностью прочитает HTML и построит объектную модель документа (DOM). Это бывает удобно, когда мы в скрипте хотим работать с документом, и должны быть уверены, что он готов.

АТРИБУТ `async`

Атрибут `async` – поддерживается всеми браузерами, кроме IE9-. Скрипт выполняется асинхронно. То есть при обнаружении `<script async src="...">` браузер не останавливает обработку страницы, а спокойно работает дальше. Когда скрипт будет загружен – он выполнится.

То есть в таком коде первым сработает тот скрипт, который раньше загрузится:

```
<script src="./first.js" async></script>  
<script src="./second.js" async></script>
```

Иногда это может быть `first.js`, иногда `second.js`. Особенно разница может быть ощутима при существенных различиях в размерах файлов и если они расположены на разных серверах. Так как порядок выполнения скриптов не гарантирован, нельзя подключать с `async` скрипты, от которых зависят какие-либо другие скрипты.

ВОПРОСЫ

- Сколько есть способов подключить JavaScript к странице, включая способы с разным порядком выполнения?
- Скрипты, подключенные с атрибутом `async` выполняются сразу же, как будут загружены браузером. Когда происходит выполнение скриптов с атрибутом `defer`?

РЕЗЮМИРУЕМ

- Есть 2 способа подключить JavaScript к странице и еще 2 способа сделать его подключение синхронным и асинхронным;
- Скрипты с атрибутом `async` выполняются в тот же момент, когда полностью будут загружены браузером;
- Скрипты с атрибутом `defer` будут выполняться в том же порядке, в котором расположены в html, но только после того, как браузер построит DOM.



РАБОТА С АТТРИБУТАМИ HTML-ЭЛЕМЕНТОВ

DOM В JAVASCRIPT

Как мы упомянули раньше, у JavaScript в браузере есть возможность обращаться к DOM.

И для этого в JavaScript зарезервированы некоторые глобальные переменные, которые можно использовать в любом месте JavaScript, но которые нельзя перезаписать или удалить.

Для обращения непосредственно к DOM существует глобальная переменная `document`, которая помогает как прочитать DOM, так и изменить его прямо на странице.

Остается вопрос, как же получить для конкретного тега соответствующий ему объект в JavaScript. Для этого существует множество способов, о которых мы расскажем на этом курсе. Но для начала воспользуемся самым простым — найдем тег по его атрибуту `id`. Для этого существует специальный метод объекта `document` — `document.getElementById()`.

В качестве примера, найдем элемент с идентификатором `main`:

```
1 <body>
2   <p id="main">
3     Текст в нашем параграфе
4   </p>
5   <script>
6     const elementMain = document.getElementById("main");
7   </script>
8 </body>
```

Иногда найти нужный объект-узел не удастся и `document.getElementById()` возвращает `null`. Это может случиться по двум причинам: либо тега с искомым атрибутом `id` нет в документе, либо не совпадает регистр символов (`getElementById()` регистрозависимый метод):

```
1 <body>
2   <p id="main">
3     Текст в нашем параграфе
4   </p>
5   <script>
6     const elementMain = document.getElementById("MAIN");
7     console.log(elementMain) // null
8
9     const elementDiv = document.getElementById("div");
10    console.log(elementDiv) // null
11  </script>
12 </body>
```

Есть немаловажный нюанс. Для каждого элемента с атрибутом `id` браузер создает одноименную глобальную переменную, к которой можно получить доступ.

```
1 <body>
2   <p id="main">
3     Текст в нашем параграфе
4   </p>
5   <script>
6     console.log(main);
7     // будет содержать параграф с одноименным "id"
8   </script>
9 </body>
```

Но! Негласно запрещается использовать такой метод для работы с элементами DOM по двум веским причинам:

1. Другой разработчик сойдет с ума в попытках найти то место, где же вы все-таки объявили эту переменную;
2. Глобальные переменные, связанные браузером по `id`, можно легко перезаписать в любом месте кода. После чего она перестанет содержать изначальный элемент.

```
1 <body>
2   <p id="main">
3     Текст в нашем параграфе
4   </p>
5   <script>
6     console.log(main);
7     // main будет содержать параграф с одноименным "id"
8     main = "Ну погоди!"
9     // main теперь содержит только "Ну погоди!"
10  </script>
11 </body>
```

Этот способ работы с элементами лишь осколок прошлого, который нужен для совместимости со старыми браузерами.

РАБОТА С АТТРИБУТАМИ HTML-ЭЛЕМЕНТОВ

Мы вспомнили про замечательный атрибут `id`, с помощью которого можно искать теги в документе. Атрибут `id` может быть применен к любому тегу. Но есть и атрибуты, которые применяются только к определенным тегам. Например, у тега `` есть специальные атрибуты: `src` — адрес картинки, `width` и `height` — ширина и высота.

```

```

А что, если нам нужно поменять какой-то атрибут во время исполнения скрипта? Это можно сделать из JavaScript, потому что все атрибуты тега являются свойствами соответствующего объекта-узла. И мы можем их прочесть и изменить. Например, можно изменить размеры картинки на странице:

```
1 
2
3 <script>
4     const image = document.getElementById("heart");
5     image.width = 100;
6     image.height = 100;
7     // Атрибуты картинки изменятся,
8     // как и её размер на странице
9 </script>
```

РАБОТА С КЛАССАМИ

Несмотря на то, что атрибут `class` является стандартным атрибутом элементов, получить его значение или изменить его напрямую через свойство `element.class` не получится.

Для работы с классами можно использовать свойство `className`:

```
1 <p id="paragraph" class="paragraph super">
2   Текст параграфа
3 </p>
4
5 <script>
6   const paragraph = document.getElementById("paragraph");
7
8   const oldClass = paragraph.className;
9   // oldClass будет равен "paragraph super"
10
11   paragraph.className = "paragraph not-super";
12   // теперь наш html-элемент будет содержать
13   // в атрибуте `class` новое значение
14 </script>
```

Это не самый удобный способ для работы с классами, поэтому на следующих уроках вы узнаете еще один способ.

СВОЙСТВО `textContent`

`textContent` позволяет получить все текстовые узлы, которые находятся внутри выбранного элемента и его потомков.

```
1 <p id="paragraph">
2   Текст <span>параграфа</span>
3 </p>
4
5 <script>
6   const paragraph = document.getElementById("paragraph");
7   console.log(paragraph.textContent); // "Текст параграфа"
8 </script>
```


Также можно записать новое значение в свойство `textContent`, но в этом случае будет удалено все содержимое элемента и заменено на текст, что мы присвоили:

```
1  <p id="paragraph">
2    Текст <span>параграфа</span>
3  </p>
4
5  <script>
6    const paragraph = document.getElementById("paragraph");
7    paragraph.textContent = "Текст параграфа";
8    console.log(paragraph.textContent);
9    // Получим также "Текст параграфа",
10   // но html уже будет содержать лишь:
11   // <p id="paragraph">Текст параграфа</p>
12 </script>
```



ВОПРОСЫ

- Где в коде можно вызвать `document` ?
- Сколько способов получения элемента из DOM вы узнали?

РЕЗЮМИРУЕМ РАБОТУ С DOM

- `document` – глобальная переменная, позволяющая работать с DOM;
- `document.getElementById()` – позволяет получить любой элемент страницы по его атрибуту `id`;
- Стандартные атрибуты элементов, вроде `id`, `src`, `href`, `title` можно получить через одноименные свойства;
- Отрисовывает макет страницы, устанавливая ширину `viewport` в качестве 100% ширины для `body`;
- Для работы с классами можно использовать свойство `className`;
- Чтобы получить текст из элемента или заменить его содержимое – используем свойство `textContent`.

ВЫЗОВ ФУНКЦИЙ ПОСЛЕ ДЕЙСТВИЯ ПОЛЬЗОВАТЕЛЯ НА СТРАНИЦЕ

СОБЫТИЯ

Приложение в браузере чаще всего построено на взаимодействии с пользователем. Человек прокручивает страницу вниз и скрипт загружает новые записи. Пользователь нажимает на кнопку и появляется выпадающее меню.

С точки зрения JavaScript такое взаимодействие построено на основе событий и обработчиков этих событий.

Существует несколько различных способов установки связи между событиями и функциями-обработчиками. Сегодня мы познакомимся с наиболее простым вариантом установки обработчиков событий — на основе свойств объектов-узлов.

Для каждого события на объектах-узлах есть соответствующие свойства, начинающиеся на `on`. Если записать в это свойство функцию, то она будет вызвана в момент наступления события.

Давайте рассмотрим эту систему на основе одного из самых распространенных событий — `click`. Событие `click` — нажатие левой кнопкой мыши — может быть применимо к любому элементу HTML документа: `<body>`, `<div>`, ``, `<button>`; и так далее. Для этого события у объекта-узла есть свойство `onclick` (по умолчанию равно `null`), в которое мы и будем записывать функцию-обработчик.

```
1 <button id="elementId">Нажми на меня</button>
2 <script>
3   const element = document.getElementById("elementId");
4   element.onclick = function() {
5     console.log('Клик!')
6   };
7 </script>
```

Значение свойства `onclick` по умолчанию равно `null`.

```
const element = document.getElementById("elementId");  
element.onclick; // null
```


Давайте рассмотрим пример с изменением размеров картинки на основе события `click`.

```
1 
2 <script>
3   const image = document.getElementById("heart");
4
5   function changeSizes() {
6     image.width = 100;
7     image.height = 100;
8   };
9
10  img.onclick = changeSizes;
11 </script>
```

Никто не мешает функцию-обработчик назначить для нескольких объектов-узлов. Стоит отметить, что для этого случая не подходят стрелочные функции, потому что контекст вызова для них недоступен.

И напоследок о том, как прекратить обработку события. Для этого в соответствующее свойство необходимо записать `null`. Например, можно отменить обработку события после первого срабатывания:

```
1 <button id="elementId">Нажми на меня</button>
2 <script>
3   const element = document.getElementById("elementId");
4
5   element.onclick = function() {
6     // Первое и единственное срабатывание
7     element.onclick = null;
8   };
9 </script>
```

Также для того, чтобы отменить стандартное действие браузера, как, например, открытие ссылки в новой вкладке при клике на элемент `<a>`, в конце события мы можем вернуть `false`.

```
1 <a href="//netology.ru" id="link">Нажми на меня</button>
2 <script>
3     const element = document.getElementById("link");
4
5     element.onclick = function() {
6         // Любой наш код при клике
7         return false;
8         // Из-за "return false" перехода
9         // по ссылке не произойдет
10    };
11 </script>
```

РЕЗЮМИРУЕМ РАБОТУ С СОБЫТИЯМИ

- В свойство `onclick` записываем функцию, которая должна быть исполнена при клике на элемент;
- Одну и ту же функцию можно использовать на события разных элементов;
- Через свойство `onclick` можно прикрепить только одну функцию;
- При помощи `return false` можно отменить стандартное действие браузера при клике на элемент.

ГЛОБАЛЬНЫЙ ОБЪЕКТ WINDOW

Для любого окружения, в котором выполняется JavaScript код, в соответствии со спецификацией ([ecma-262](#)) необходимо наличие глобального объекта.

В браузере глобальный объект — это `window`. Любые переменные, объявленные с помощью `var` вне какой-либо функции, являются свойствами глобального объекта `window`. Например:

```
1  var name = "Иван";
2  window.name; // будет равно "Иван"
3  // или
4  window.lastname = "Иванов";
5  console.log(lastname); // будет равно "Иванов"
```

Другой эффект при использовании одноименных свойств в `window` и переменной `const` или `let`:

```
1  const age = 1;  
2  window.age = 99;  
3  console.log(age); // будет равно 1
```

НЕМНОГО ПРО ОБЪЕКТНУЮ МОДЕЛЬ БРАУЗЕРА

ВВЕДЕНИЕ В BOM

Мы уже знаем, что в браузере для работы с HTML реализована модель DOM с главным объектом `document` для взаимодействия с HTML-элементами страницы. Но есть другая модель — **BOM**.

BOM — объектная модель браузера, то есть все те объекты, с помощью которых можно взаимодействовать непосредственно с браузером.

И в качестве первых объектов BOM мы напомним вам о функциях-таймерах `setTimeout` и `setInterval`.

Давайте напишем небольшой скрипт, который будет каждую секунду увеличивать число в абзаце на два:

```
1 <p id="output">1</p>
2
3 <script>
4   const addText = function(){
5     const output = document.getElementById("output");
6     output.textContent *= 2;
7   }
8
9   setInterval(addText, 1000)
10 </script>
```

РЕЗЮМИРУЕМ ТО, ЧТО УЗНАЛИ ПО BOM

- BOM — объектная модель браузера; содержит множество вспомогательных функций;
- `setTimeout` — позволяет отложить выполнение функции на N миллисекунд;
- `setInterval` — позволяет запускать функцию каждые N миллисекунд;
- `setTimeout` и `setInterval` могут быть отменены через `clearTimeout` и `clearInterval` соответственно.

ФИНАЛЬНОЕ РЕЗЮМЕ

- Браузер имеет в своей основе объект window, который включает в себя три основополагающих элемента: DOM, BOM, JavaScript;
- DOM — позволяет работать с документом HTML;
- BOM — набор вспомогательных методов и функций, используемых для работы с браузером из JavaScript;
- JavaScript — следует за стандартами, а поскольку разные браузеры вводят стандарты с разной скоростью — в разных браузерах могут быть недоступны те или иные возможности языка.

МАТЕРИАЛЫ, ИСПОЛЬЗОВАННЫЕ ПРИ ПОДГОТОВКЕ ЛЕКЦИИ

- <https://developers.google.com/web/fundamentals/performance/critical-rendering-path>
- <https://learn.JavaScript.ru/browser-environment>
- <https://learn.JavaScript.ru/dom-nodes>
- <https://habr.com/ru/post/320430/>
- <https://learn.JavaScript.ru/external-script>

ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаем в группе Facebook!
- Работы должны соответствовать принятому [стилю оформления кода](#).
- Зачет по домашней работе проставляется после того, как приняты все **3 задачи**.



Ваши вопросы?

ВЛАДИМИР ЯЗЫКОВ