Возможности JavaScript в браузере

Полина Шнайдер
Веб-разработчик

Полина Шнайдер

Веб-разработчик

Что вас ждет 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

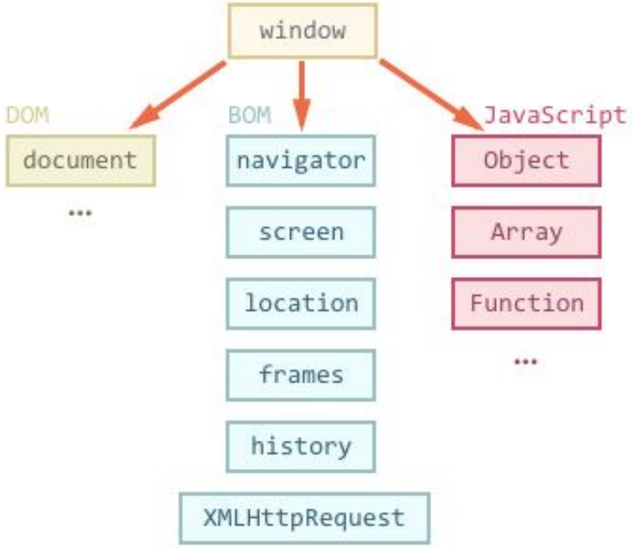
1

Браузеры

Браузер — это программное обеспечение для отображения веб-страниц.

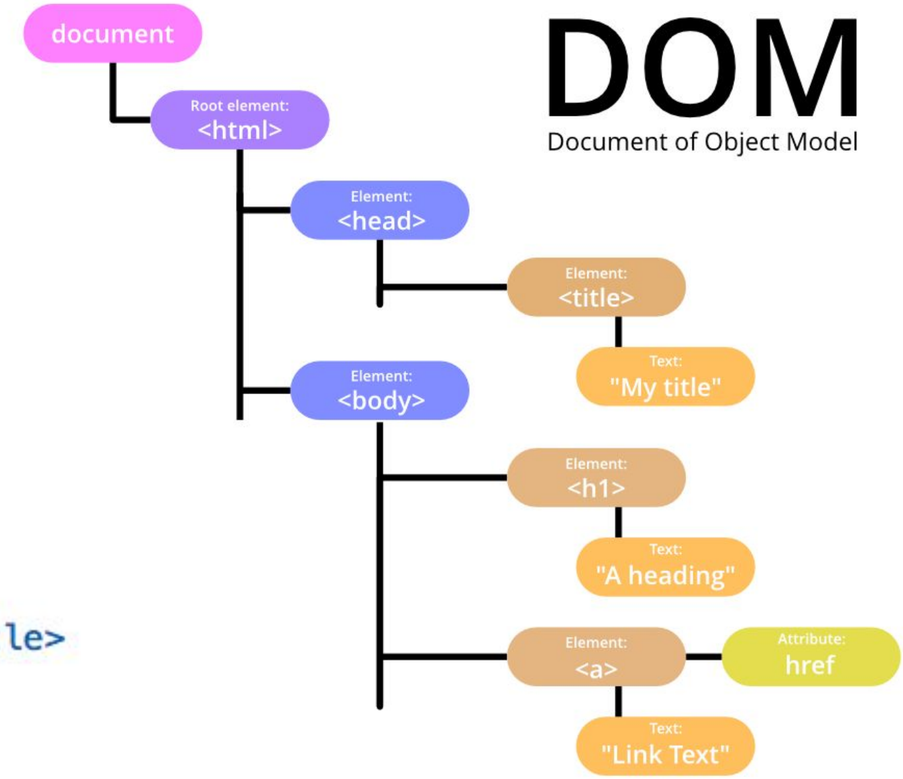
Каждый браузер имеет объект window и **три** важных элемента в его структуре:

- 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 трем остальным элементам: window, DOM и BOM



DOM (Document Object Model)

Это объектная модель документа, которая представляет все содержимое страницы в виде объектов, которые можно менять.




```
<!DOCTYPE html>
    <html>

        <head>
            <title> Title here </title>
        </head>

        <body>
            Web page content goes here.
        </body>

    </html>
```

BOM (Browser Object Model)

Это дополнительные объекты, предоставляемые браузером (окружением), чтобы работать со всем, кроме документа

Например:

- Объект **navigator** дает информацию о самом браузере и операционной системе
- Объект **location** позволяет получить текущий URL и перенаправить браузер по новому адресу.

Далее подробнее

Как подключить JavaScript на страницу

2

Подключаем JavaScript

Давайте посмотрим на очень простую HTML-страницу:

```
<html>
  <head>
    <title>Заголовок</title>
  </head>
  <body>
    Привет, Мир!
  </body>
</html>
```

В ней нет ничего, связанного с **JavaScript**, поэтому браузер просто отобразит строку **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:

```
<html>
  <head>
    <title>Заголовок</title>
  </head>
  <body>
    Привет
    <script>
        console.log("Мир!");
    </script>
  </body>
</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.

Подключить JavaScript на страницу можно двумя способами

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.

Подключить JavaScript на страницу можно двумя способами

```
<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`.

Подключить JavaScript на страницу можно двумя способами

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

3

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

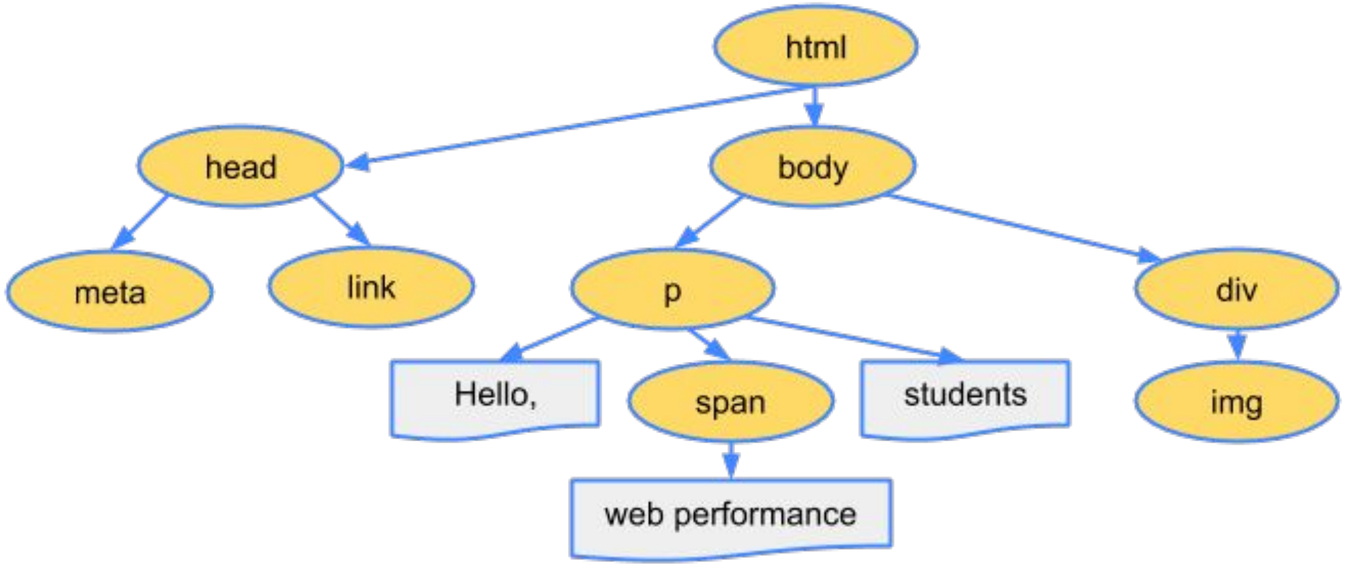
4

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:

```
<html>
  <head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>
    <p>
      Hello <span>web perfomance</span> students
    </p>
    <div>
      <img src="image.jpg">
    </div>
  </body>
</html>
```

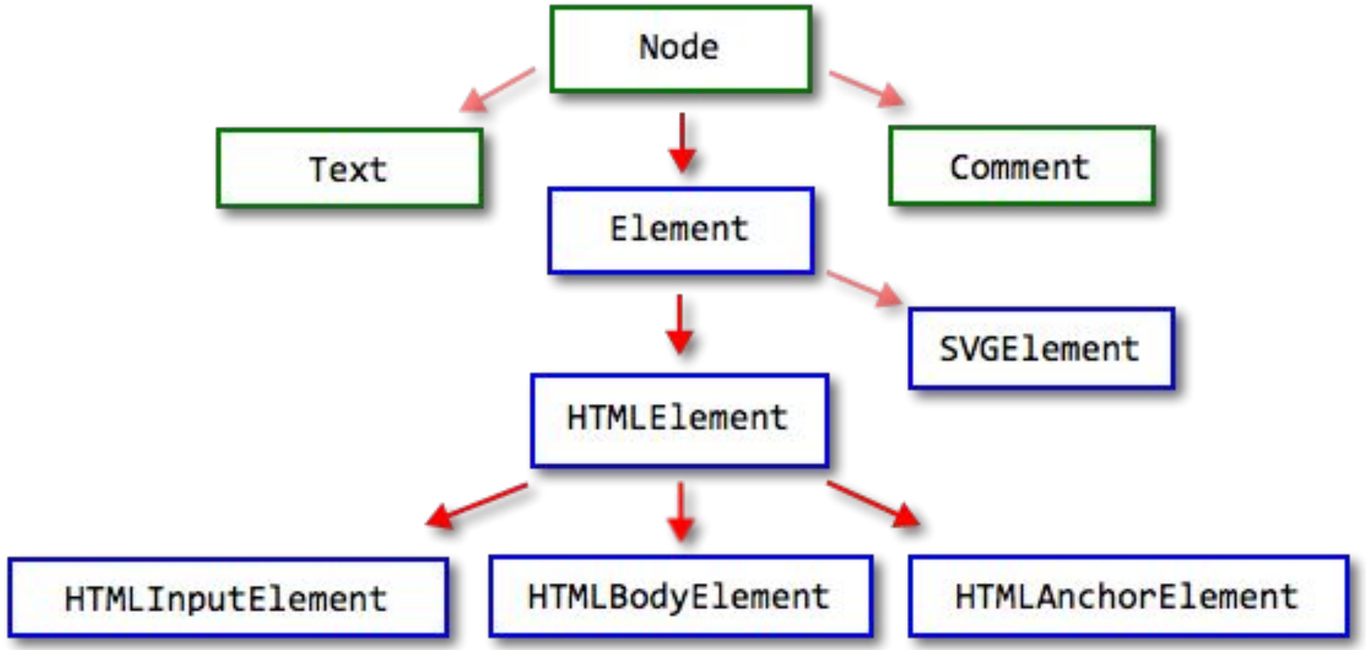


Как вы видите, иерархия DOM полностью повторяет иерархию html. Но состоит она из узлов и соединений, которые указывают на потомков конкретного узла. Узлами в данном случае выступают теги (желтые блоки) и текстовое содержимое этих тегов (голубые блоки). Или, по-другому, это *узлы-элементы* и *текстовые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



Немного про узлы

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`.

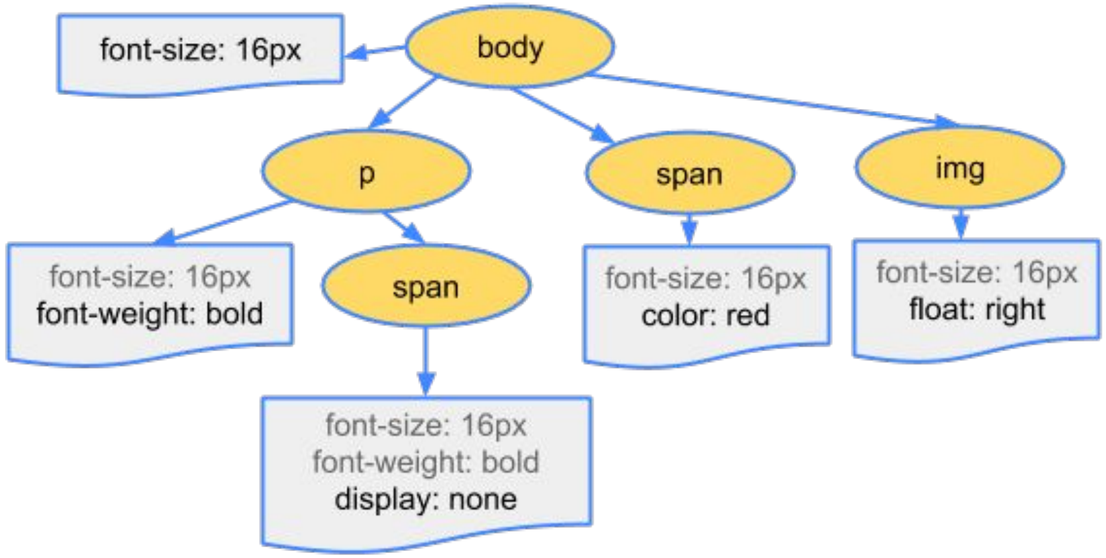
И третий, основной тип узла (Comment) — обычный HTML комментарий.

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):

```
body { font-size: 16px }
p { font-weight: bold }
span { color: red }
p span { display: none }
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.

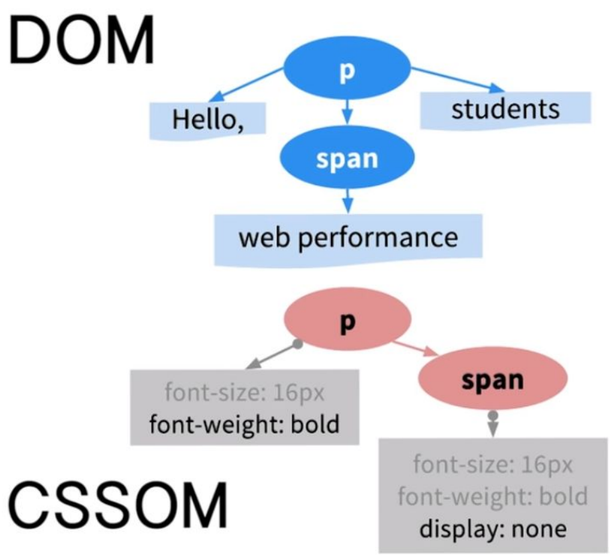
Обратите внимание, что данная схема из примера отображает только загруженные стили без стандартных стилей браузера.

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

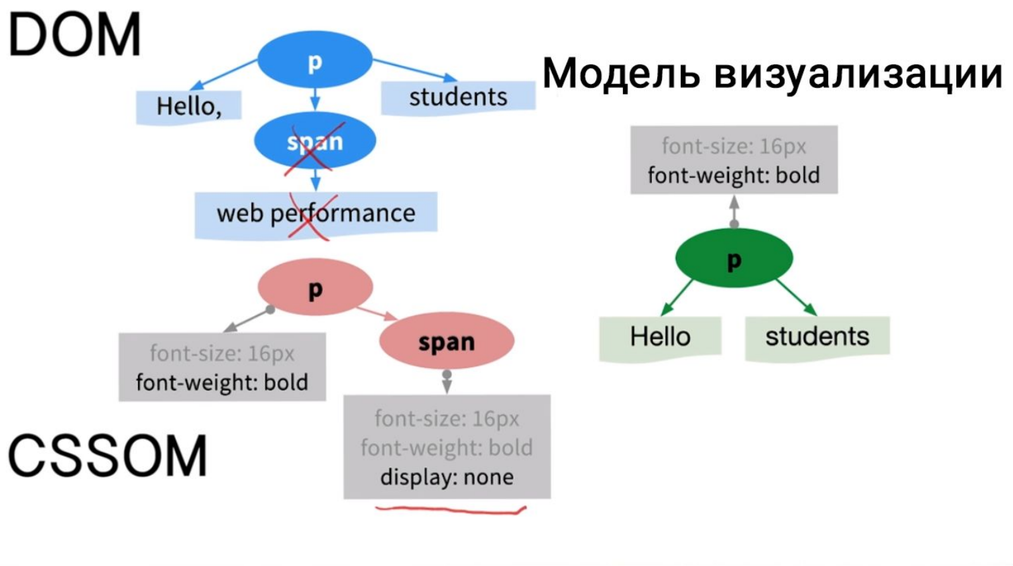
После построения DOM и CSSOM браузер приступает к **объединению** этих двух моделей, чтобы создать общую структуру, по которой можно будет начать отрисовывать сам макет страницы. Эта объединенная модель называется **моделью визуализации**.

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



Модель визуализации

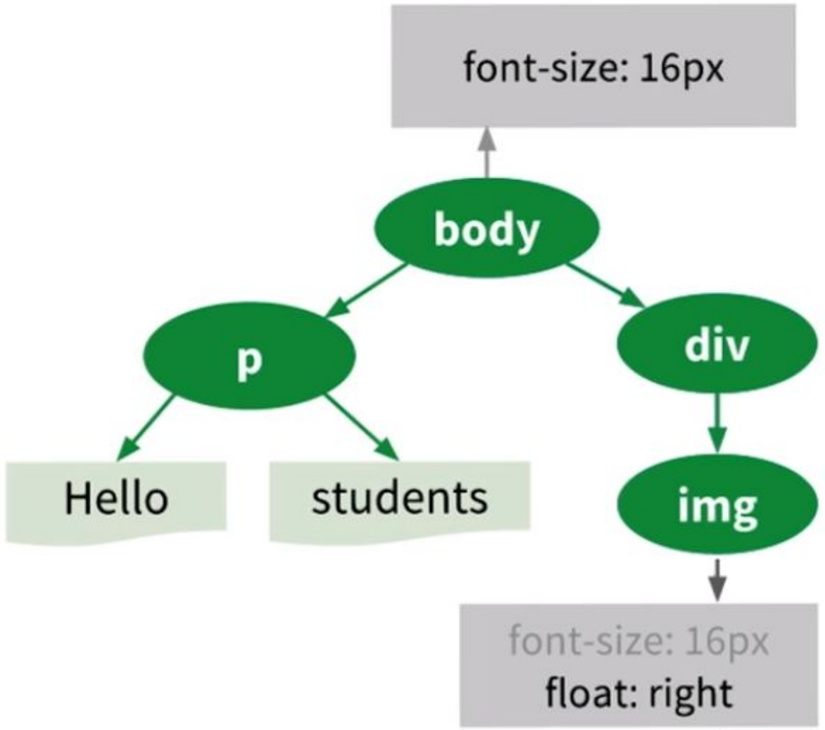
Объединив эти модели, мы получим следующую модель визуализации:



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.

Модель визуализации

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%`.

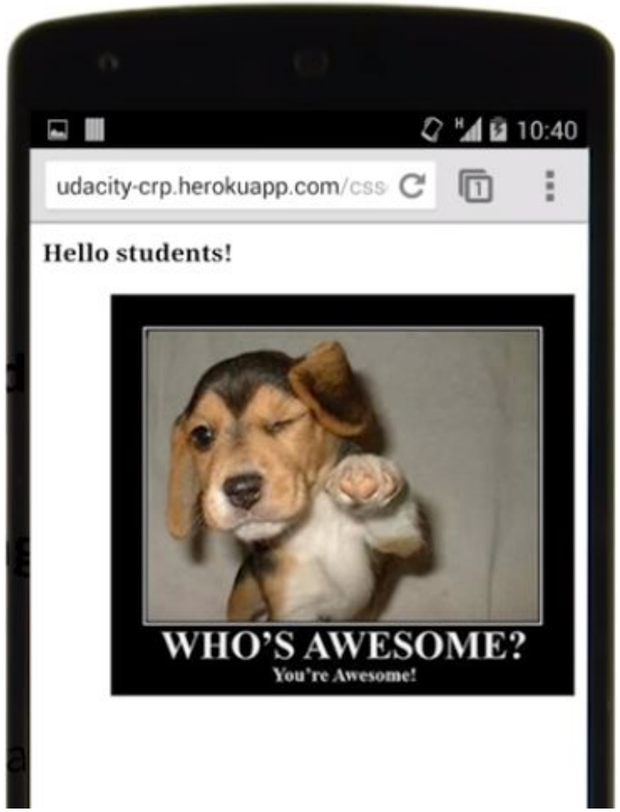
Если все потомки тега body берут за `100%` ширину от своего родителя, то тег body за `100%` берет ширину от `viewport`. Обычно она устанавливается через тег:

```
<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:



Вопрос

- В каком порядке браузер производит расчет следующих элементов:
  1 – CSSOM, 2 – Модель визуализации, 3 – Макет страницы, 4 – DOM?

Напишите **последовательность** цифр в чат

Ответ

- В каком порядке браузер производит расчет следующих элементов:
  1 – CSSOM, 2 – Модель визуализации, 3 – Макет страницы, 4 – DOM?

Напишите **последовательность** цифр в чат

4123

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

5

Синхронное и асинхронное выполнение JavaScript

Мы уже знаем, как подключается JavaScript на страницу, но нужно остановится 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.

```
<body>
  <p>Hello</p>
  <script src="script.js">
  <!--
    Все, что идет далее, будет выведено на страницу
    только после выполнения script.js
  -->
  <p>students!</p>
</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`.

Атрибут `defer`

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.

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 атрибутами html-элементов

6

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.

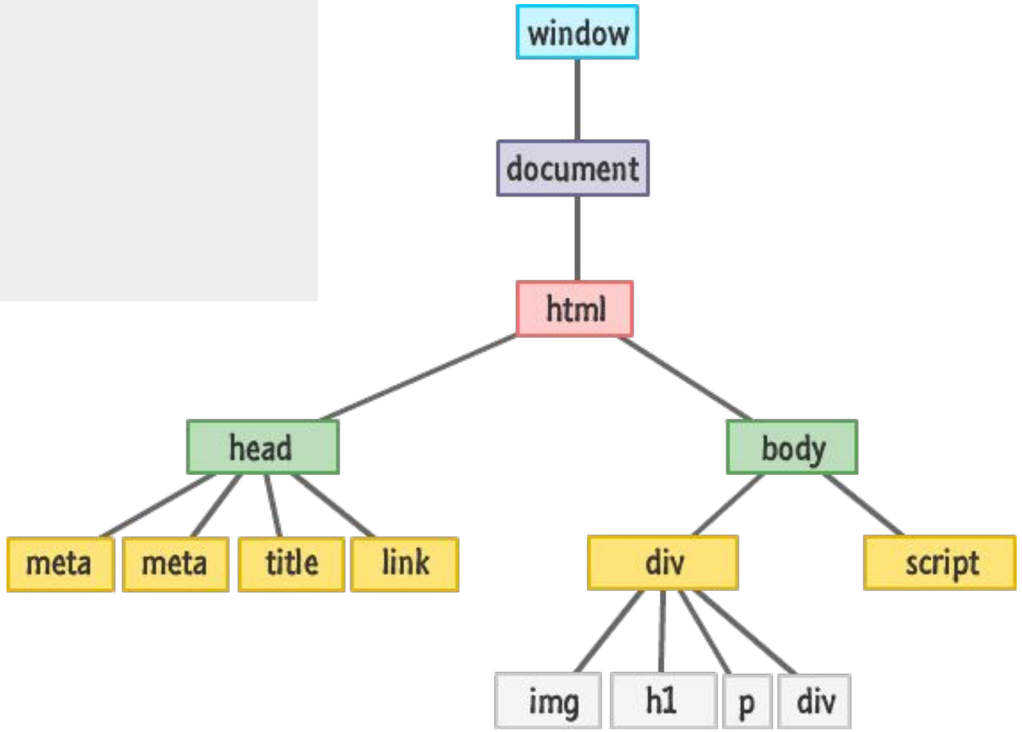
Для обращения непосредственно к DOM существует глобальная переменная **document**, которая помогает как прочитать DOM, так и изменить его прямо на странице.

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()`.

DOM в JavaScript

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

```
<body>
  <p id="main">
    Текст в нашем параграфе
  </p>
  <script>
    const elementMain = document.getElementById("main");
  </script>
</body>
```



DOM в JavaScript

Иногда найти нужный объект-узел **не удается** 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):

```
<body>
  <p id="main">
    Текст в нашем параграфе
  </p>
  <script>
    const elementMain = document.getElementById("MAIN");
    console.log(elementMain) // null

    const elementDiv = document.getElementById("div");
    console.log(elementDiv) // null
  </script>
</body>
```

DOM в JavaScript

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

```
<body>
  <p id="main">
    Текст в нашем параграфе
  </p>
  <script>
    console.log(main);
    // будет содержать параграф с одноименным "id"
  </script>
</body>
```

DOM в JavaScript

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.

```
<body>
  <p id="main">
    Текст в нашем параграфе
  </p>
  <script>
    console.log(main);
    // main будет содержать параграф с одноименным "id"
    main = "Ну погоди!"
    // main теперь содержит  только "Ну погоди!"
  </script>
</body>
```

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

Работа с атрибутами html-элементов

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

```
<img id="heart" src="http://bit.ly/2nyH0tY" width="150" height="150">
```

Работа с атрибутами html-элементов

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 прочитать и изменить. Например, можно изменить размеры картинки на странице:

```
<img id="heart" src="http://bit.ly/2nyH0tY" width="150" height="150">

<script>
    const image = document.getElementById("heart");
    image.width = 100;
    image.height = 100;
    // Атрибуты картинки изменятся,
    // как и её размер на странице
</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`:

```
<p id="paragraph" class="paragraph super">
  Текст параграфа
</p>

<script>
  const paragraph = document.getElementById("paragraph");

  const oldClass = paragraph.className;
  // oldClass будет равен "paragraph super"

  paragraph.className = "paragraph not-super";
  // теперь наш html-элемент будет содержать
  // в атрибуте `class` новое значение
</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.

```
<p id="paragraph">
  Текст <span>параграфа</span>
</p>

<script>
  const paragraph = document.getElementById("paragraph");
  console.log(paragraph.textContent); // "Текст параграфа"
</script>
```

Свойство `textContent`

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

```
<p id="paragraph">
  Текст <span>параграфа</span>
</p>

<script>
  const paragraph = document.getElementById("paragraph");
  paragraph.textContent = "Текст параграфа";
  console.log(paragraph.textContent);
  // Получим также "Текст параграфа",
  // но html уже будет содержать лишь:
  // <p id="paragraph">Текст параграфа</p>
</script>
```

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;
- 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

7

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.

События

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.

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>`, `<img>`, `<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.

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

События

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

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

События

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

```
<img id="heart" src="http://bit.ly/2nyH0tY" width="150" height="150">
<script>
  const image = document.getElementById("heart");

  function changeSizes() {
    image.width = 100;
    image.height = 100;
  };

  img.onclick = changeSizes;
</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.

События

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

```
<button id="elementId">Нажми на меня</button>
<script>
  const element = document.getElementById("elementId");

  element.onclick = function() {
    // Первое и единственное срабатывание
    element.onclick = null;
  };
</script>
```

События

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`.

```
<a href="//netology.ru" id="link">Нажми на меня</button>
  <script>
    const element = document.getElementById("link");

    element.onclick = function() {
      // Любой наш код при клике
      return false;
      // Из-за "return false" перехода
      // по ссылке не произойдет
    };
  </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 исполняется 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:

```
var name = "Иван";
window.name; // будет равно "Иван"
// или
window.lastname = "Иванов";
console.log(lastname); // будет равно "Иванов"
```

Глобальный объект `window`

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

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

Немного про объектную модель браузера

8

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`.

Введение в BOM

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

```
<p id="output">1</p>

<script>
  const addText = function(){
    const output = document.getElementById("output");
    output.textContent *= 2;
  }

  setInterval(addText, 1000)
</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

1. https://developers.google.com/web/fundamentals/performance/critical-rendering-path
2. https://learn.JavaScript.ru/browser-environment
3. https://learn.JavaScript.ru/dom-nodes
4. https://habr.com/ru/post/320430/
5. https://learn.javascript.ru/script-async-defer

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.

Задавайте вопросы и напишите отзыв о лекции!

Полина Шнайдер
Веб-разработчик