

10 - Объектная модель документа (DOM)

1. Работа с документами

Клиентский JavaScript предназначен для того, чтобы превращать статические HTML-документы в интерактивные веб-приложения. Работа с содержимым веб-страниц – главное предназначение JavaScript.

Объектная модель документа (Document Object Model, DOM) – это фундаментальный прикладной программный интерфейс, обеспечивающий возможность работы с содержимым HTML- и XML-документов. Прикладной программный интерфейс (API) модели DOM не особенно сложен, но в нем существует множество архитектурных особенностей, которые вы должны знать. DOM – это представление документа в виде дерева объектов, доступное для изменения через JavaScript. В этом дереве выделено два типа узлов.

1. **Теги** образуют узлы-элементы (Element). Естественным образом одни узлы вложены в другие. Структура дерева образована исключительно за счет них.
2. **Текст** внутри элементов образует текстовые узлы (Text), обозначенные как. Текстовый узел содержит исключительно строку текста и не может иметь потомков, то есть он всегда на самом нижнем уровне.
3. **Комментарии** – иногда в них можно включить информацию, которая не будет показана, но доступна из JS.

Корнем дерева является узел Document, который представляет документ целиком. Узлы, представляющие HTML-элементы, являются узлами типа Element, а узлы, представляющие текст, – узлами типа Text.

Тем, кто еще не знаком с древовидными структурами в компьютерном программировании, полезно узнать, что терминология для их описания была заимствована у генеалогических деревьев. Узел, расположенный непосредственно над данным узлом, называется **родительским** по отношению к данному узлу. Узлы, расположенные на один уровень ниже другого узла, являются **дочерними** по отношению к данному узлу. Узлы, находящиеся на том же уровне и имеющие того же родителя, называются **братьями**. Узлы, расположенные на любое число уровней ниже другого узла, являются его **потомками**. Родительские, прародительские и любые другие узлы, расположенные выше данного узла, являются его **предками**.

2. Выбор элементов документа

Работа большинства клиентских программ на языке JavaScript так или иначе связана с манипулированием элементами документа. В ходе выполнения эти программы могут использовать глобальную переменную document. Однако, чтобы выполнить какие-либо манипуляции с элементами документа, программа должна каким-то образом получить, или выбрать, объекты Element, ссылающиеся на эти элементы документа. Модель DOM определяет несколько способов выборки элементов. Выбрать элемент или элементы документа можно:

2.1. getElementById() - выбор элементов по значению атрибута id. Все HTML-элементы имеют атрибуты id. Значение этого атрибута должно быть уникальным в

пределах документа – никакие два элемента в одном и том же документе не должны иметь одинаковые значения атрибута id.

```
var section1 = document.getElementById("section1");
```

2.2. `getElementsByName()` - выбор элементов по значению атрибута `name`.

HTML-атрибут `name` первоначально предназначался для присваивания имен элементам форм, и значение этого атрибута использовалось, когда выполнялась отправка данных формы на сервер. Подобно атрибуту `id`, атрибут `name` присваивает имя элементу. Однако, в отличие от `id`, значение атрибута `name` не обязано быть уникальным: одно и то же имя могут иметь сразу несколько элементов, что вполне обычно при использовании в формах радиокнопок и флажков. Кроме того, в отличие от `id`, атрибут `name` допускается указывать лишь в некоторых HTML-элементах, включая формы, элементы форм и элементы `<iframe>` и ``.

```
var radiobuttons = document.getElementsByName("favorite_color");
```

2.3. `getElementsByTagName()` - выбор элементов по типу. Метод `getElementsByTagName()` объекта `Document` позволяет выбрать все HTML- или XML-элементы указанного типа (или по имени тега). Например, получить подобный массиву объект, доступный только для чтения, содержащий объекты `Element` всех элементов `` в документе, можно следующим образом:

```
var spans = document.getElementsByTagName("span");
```

2.4. `getElementsByClassName()` - выбор элементов по классу CSS. Значением HTML-атрибута `class` является список из нуля или более идентификаторов, разделенных пробелами. Он дает возможность определять множества связанных элементов документа: любые элементы, имеющие в атрибуте `class` один и тот же идентификатор, являются частью одного множества. Слово `class` зарезервировано в языке JavaScript, поэтому для хранения значения HTML-атрибута `class` в клиентском JavaScript используется свойство `className`. стандарт HTML5 определяет метод `getElementsByClassName()`, позволяющий выбирать множества элементов документа на основе идентификаторов в их атрибутах `class`:

```
// Отыскать все элементы с идентификатором  
"warning" в атрибуте class
```

```
var warnings = document.getElementsByClassName("warning");
```

```
// Отыскать всех потомков элемента с  
именем "log" с идентификаторами "error"  
// и "fatal" в атрибуте class
```

```
var log = document.getElementById("log");
```

```
var fatal = log.getElementsByClassName("fatal error");
```

2.5. `querySelectorAll()` - выбор элементов с использованием селекторов CSS.

Каскадные таблицы стилей CSS имеют очень мощные синтаксические конструкции, известные как селекторы, позволяющие описывать элементы или множества элементов документа. Элементы можно описать с помощью имени тега и атрибутов `id` и `class`:

```
#nav // Элемент с атрибутом id="nav"  
div // Любой элемент <div>
```

```
.warning // Люб о й э л е м е н т с и д е н т и ф и к а т о р о м  
"warning" в а т р и б у т е c l a s s
```

В более общем случае элементы можно выбирать, опираясь на значения атрибутов:

```
p[lang="fr"] // А б з а ц с т е к с т о м н а ф р а н ц у з с к о м  
я з ы к е : <p lang="fr">  
*[name="x"] // Люб о й э л е м е н т с а т р и б у т о м n a m e = " x "
```

Эти простейшие селекторы можно комбинировать:

```
span.fatal.error // Люб о й э л е м е н т <span> с к л а с с а м и  
"fatal" и "error"  
span[lang="fr"].warning // Люб о е п р е д у п р е ж д е н и е н а  
ф р а н ц у з с к о м я з ы к е
```

С помощью селекторов можно также определять взаимоотношения между элементами:

```
#log span // Люб о й <span>, я в л я ю щ и й с я п о т о м к о м  
э л е м е н т а с i d = " l o g "  
#log>span // Люб о й <span>, д о ч е р н и й п о о т н о ш е н и ю к  
э л е м е н т у с i d = " l o g "  
body>h1:first-child // П е р в ы й <h1>, д о ч е р н и й п о  
о т н о ш е н и ю к <body>
```

Селекторы можно комбинировать для выбора нескольких элементов или множеств элементов:

```
div, #log // В с е э л е м е н т ы <div> п л ю с э л е м е н т с  
i d = " l o g "
```

Как видите, селекторы CSS позволяют выбирать элементы всеми способами, описанными выше: по значению атрибута id и name, по имени тега и по имени класса.

В дополнение к методу `querySelectorAll()` объект документа также определяет метод **`querySelector()`**, подобный методу `querySelectorAll()`, – с тем отличием, что он возвращает только первый (в порядке следования в документе) соответствующий элемент или `null`, в случае отсутствия соответствующих элементов.

3. Структура документа и навигация по документу.

После выбора элемента документа иногда бывает необходимо отыскать структурно связанные части документа (родитель, братья, дочерний элемент).

Документы как деревья элементов. Когда основным интерес представляют сами элементы документа, а не текст в них (и пробельные символы между ними), гораздо удобнее использовать прикладной интерфейс, позволяющий интерпретировать документ как дерево объектов `Element`, игнорируя узлы `Text` и `Comment`, которые также являются частью документа.

Свойство children объектов Element подобно свойству childNodes, его значением является объект NodeList. Однако, в отличие от свойства childNodes, список children содержит только объекты Element.

Свойство parentElement - родительский узел данного узла или null для узлов, не имеющих родителя, таких как Document.

Второй частью прикладного интерфейса навигации по элементам документа являются свойства объекта Element, аналогичные свойствам доступа к дочерним и братским узлам объекта Node:

firstElementChild, lastElementChild - Похожи на свойства firstChild и lastChild, но возвращают дочерние элементы.

nextElementSibling, previousElementSibling - Похожи на свойства nextSibling и previousSibling, но возвращают братские элементы.

childElementCount - Количество дочерних элементов. Возвращает то же значение, что и свойство children.length.

Эти свойства доступа к дочерним и братским элементам стандартизованы и реализованы во всех текущих браузерах.

4. Атрибуты

HTML-элементы состоят из имени тега и множества пар имя/значение, известных как атрибуты. Например, элемент <a>, определяющий гиперссылку, в качестве адреса назначения ссылки использует значение атрибута href. Значения атрибутов HTML-элементов доступны в виде свойств объектов HTMLElement, представляющих эти элементы.

HTML-атрибуты как свойства объектов Element. Объекты HTMLElement, представляющие элементы HTML-документа, определяют свойства, доступные для чтения/записи, соответствующие HTML-атрибутам элементов. Объект HTMLElement определяет свойства для поддержки универсальных HTTP-атрибутов, таких как id, title, lang и dir, и даже свойства-обработчики событий, такие как onclick. Специализированные подклассы класса Element определяют атрибуты, характерные для представляемых ими элементов. Например, узнать URL-адрес изображения можно, обратившись к свойству src объекта HTML-Element, представляющего элемент :

```
var image = document.getElementById("myimage");
var imgurl = image.src; // Атрибут src определяет
URL-адрес изображения
image.id === "myimage" // Потому что поиск элемента
выполнялся по id
```

Имена атрибутов в разметке HTML не чувствительны к регистру символов, в отличие от имен свойств в языке JavaScript. Чтобы преобразовать имя атрибута в имя свойства в языке JavaScript, его нужно записать символами в нижнем регистре. Однако, если имя атрибута состоит из более чем одного слова, первый символ каждого слова, кроме первого, записывается в верхнем регистре, например:

defaultChecked и tabIndex.

Имена некоторых HTML-атрибутов совпадают с зарезервированными словами языка JavaScript. Имена свойств, соответствующих таким атрибутам, начинаются с приставки «html». Например, HTML-атрибуту `for` (элемента `<label>`) в языке JavaScript соответствует свойство с именем `htmlFor`. Очень важный HTML-атрибут **class**, имя которого совпадает с зарезервированным (но не используемым) в языке JavaScript словом «class», является исключением из этого правила: в программном коде на языке JavaScript ему соответствует свойство **className**.

Доступ к нестандартным HTML-атрибутам. Тип `Element` определяет дополнительные методы **getAttribute()** и **setAttribute()**, которые можно использовать для доступа к нестандартным HTML-атрибутам, а также обращаться к атрибутам элементов XML-документа:

```
var image = document.images[0];
var width = parseInt(image.getAttribute("WIDTH"));
image.setAttribute("class", "thumbnail");
```

Пример выше демонстрирует важные отличия между этими методами и прикладным интерфейсом, эти методы принимают стандартные имена атрибутов, даже если они совпадают с зарезервированными словами языка JavaScript. Класс `Element` также определяет два родственных метода, **hasAttribute()** и **removeAttribute()**. Первый из них проверяет присутствие атрибута с указанным именем, а второй удаляет атрибут. Эти методы особенно удобны при работе с логическими атрибутами: для этих атрибутов (таких как атрибут `disabled` HTML-форм) важно их наличие или отсутствие в элементе, а не их значения.

Обратите внимание, что основанный на свойствах прикладной интерфейс получения доступа к значениям атрибутов не позволяет удалять атрибуты из элементов.

Атрибуты с данными. Иногда бывает желательно добавить в HTML-элементы дополнительные данные, обычно когда предусматривается возможность выбора этих элементов в JavaScript-сценариях и выполнения некоторых операций с ними. Иногда это можно реализовать, добавив специальные идентификаторы в атрибут `class`. Иногда, когда речь заходит о более сложных данных, программисты прибегают к использованию нестандартных атрибутов. Как отмечалось выше, для чтения и изменения значений нестандартных атрибутов можно использовать методы `getAttribute()` и `setAttribute()`. Платой за это будет несоответствие документа стандарту.

Стандарт HTML5 предоставляет решение этой проблемы. В документах, соответствующих стандарту HTML5, все атрибуты, имена которых состоят только из символов в нижнем регистре и начинаются с приставки «data-», считаются допустимыми. Эти «атрибуты с данными» не оказывают влияния на представление элементов, в которых присутствуют, и обеспечивают стандартный способ включения дополнительных данных без нарушения стандартов.

Кроме того, стандарт HTML5 определяет в объекте `Element` свойство `dataset`. Это свойство ссылается на объект, который имеет свойства, имена которых соответствуют именам атрибутов `data-` без приставки. То есть свойство `dataset.x` будет хранить значение атрибута `data-x`. Имена атрибутов с дефисами отображаются в имена свойств с переменным регистром символов: атрибут `data-jquery-test` превратится в свойство `dataset.jqueryTest`.

5. Содержимое элемента.

Содержимое элемента в виде HTML. При чтении свойства **innerHTML** объекта **Element** возвращается содержимое этого элемента в виде строки разметки. Попытка изменить значение этого свойства приводит к вызову синтаксического анализатора веб-браузера и замещению текущего содержимого элемента разобранном представлением новой строки.

Веб-браузеры прекрасно справляются с синтаксическим анализом разметки HTML, поэтому операция изменения значения свойства **innerHTML** обычно достаточно эффективна, несмотря на необходимость синтаксического анализа. Тем не менее обратите внимание, что многократное добавление фрагментов текста в свойство **innerHTML** с помощью оператора **+=** обычно далеко не эффективное решение, потому что требует выполнения двух шагов – сериализации и синтаксического анализа.

Кроме того, спецификация HTML5 стандартизует свойство с именем **outerHTML**. При обращении к свойству **outerHTML** оно возвращает строку разметки HTML, содержащую открывающий и закрывающий теги элемента, которому принадлежит это свойство. При записи нового значения в свойство **outerHTML** элемента новое содержимое замещает элемент целиком.

Содержимое элемента в виде простого текста. Иногда бывает необходимо получить содержимое элемента в виде простого текста или вставить простой текст в документ (без необходимости экранировать угловые скобки и амперсанды, используемые в разметке HTML). Стандартный способ выполнения этих операций основан на использовании **свойства textContent** объекта **Node**:

```
var para = document.getElementsByTagName("p")[0]; // Первый <p> в документе
var text = para.textContent; // Текст "This is a simple document."
para.textContent = "Hello World!"; // Изменит содержимое абзаца
```

6. Создание, вставка и удаление узлов. Мы уже знаем, как получать и изменять содержимое документа, используя строки с разметкой HTML и с простым текстом. Мы также знаем, как выполнять обход документа для исследования отдельных узлов **Element** и **Text**, составляющих его содержимое. Однако точно так же существует возможность изменения документа на уровне отдельных узлов. Тип **Document** определяет методы создания объектов **Element** и **Text**, а тип **Node** определяет методы для вставки, удаления и замены узлов в дереве. Приемы создания и вставки узлов показаны в примере ниже:

```
// Асинхронная загрузка сценария из
указанного URL-адреса и его выполнение
function loadasync(url) {
    var head = document.getElementsByTagName("head")[0]; //
Отыскать <head>
    var s = document.createElement("script"); // Создать
элемент <script>
    s.src = url; // Установить его атрибут src

    head.appendChild(s); // Вставить <script> в <head>
}
```

Создание узлов. Как было показано в примере выше, создавать новые узлы Element можно с помощью метода **createElement()** объекта Document. Этому методу необходимо передать имя тега: это имя не чувствительно к регистру символов при работе с HTML-документами.

Для создания текстовых узлов существует аналогичный метод:

```
var newnode = document.createTextNode("содержимое  
текстового узла");
```

Еще один способ создания в документе новых узлов заключается в копировании существующих узлов. Каждый узел имеет метод **cloneNode()**, возвращающий новую копию узла. Если передать ему аргумент со значением true, он рекурсивно создаст копии всех потомков, в противном случае будет создана лишь поверхностная копия.

Вставка узлов. После создания нового узла его можно вставить в документ с помощью методов типа Node: **appendChild()** или **insertBefore()**. Метод **appendChild()** вызывается относительно узла Element, в который требуется вставить новый узел, и вставляет указанный узел так, что тот становится последним дочерним узлом (значением свойства **lastChild**).

Метод **insertBefore()** похож на метод **appendChild()**, но он принимает два аргумента. В первом аргументе указывается вставляемый узел, а во втором – узел, перед которым должен быть вставлен новый узел. Этот метод вызывается относительно объекта узла, который станет родителем нового узла, а во втором аргументе должен передаваться дочерний узел этого родителя. Если во втором аргументе передать null, метод **insertBefore()** будет вести себя, как **appendChild()**, и вставит узел в конец.

Ниже приводится простая функция вставки узла в позицию с указанным числовым индексом. Она демонстрирует применение обоих методов, **appendChild()** и **insertBefore()**:

```
// Вставляет узел child в узел parent так, что он  
// становится n-м дочерним узлом  
function insertAt(parent, child, n) {  
    if (n < 0 || n > parent.childNodes.length) {  
        throw new Error("недопустимый индекс");  
    } else if (n == parent.childNodes.length) {  
        parent.appendChild(child);  
    } else {  
        parent.insertBefore(child, parent.childNodes[n]);  
    }  
}
```

Если метод **appendChild()** или **insertBefore()** используется для вставки узла, который уже находится в составе документа, этот узел автоматически будет удален из текущей позиции и вставлен в новую позицию; нет необходимости явно удалять узел.

Удаление и замена узлов. Метод **removeChild()** удаляет элемент из дерева документа. Но будьте внимательны: этот метод вызывается не относительно узла, который должен быть удален, а (как следует из фрагмента «child» в его имени) относительно родителя удаляемого узла. Этот метод вызывается относительно родителя

и принимает в виде аргумента дочерний узел, который требуется удалить. Чтобы удалить узел n из документа, вызов метода должен осуществляться так:

```
n.parentNode.removeChild(n);
```

Метод **replaceChild()** удаляет один дочерний узел и замещает его другим. Этот метод должен вызываться относительно родительского узла. В первом аргументе он принимает новый узел, а во втором – замещаемый узел. Например, ниже показано, как заменить узел n текстовой строкой:

```
n.parentNode.replaceChild(document.createTextNode("[  
ИСПРАВЛЕНО ]"), n);
```

Задание на лекции:

3. Продолжить работу над калькулятором
4. Начать реализацию аналога - google keep

Домашнее задание:

1. Замена span на тег b без изменения текста. по нажатию на кнопку (она меняет span на тег b, не изменяя при этом текст внутри тега)
2. На `getElementsByTagName`. Задача. Дан HTML код (см. под задачей). Поменяйте содержимое абзацев на их порядковый номер в коде.

```
<h2>Заголовок, не поменяется.</h2>
```

```
<p>Абзац, поменяется.</p>
```

```
<p>Абзац, поменяется.</p>
```

```
<p>Абзац, поменяется.</p>
```

3. На `getElementsByClassName`. Задача. Дан HTML код (см. под задачей). Поменяйте содержимое элементов с классом zzz на их порядковый номер в коде.

```
<h2 class="zzz">Заголовок с классом zzz.</h2>
```

```
<p class="zzz">Абзац с классом zzz.</p>
```

```
<p class="zzz">Абзац с классом zzz.</p>
```

```
<p>Просто абзац, не поменяется.</p>
```