Конспект лекции 14. Понятие DOM модели. События в JavaScript

1. Что такое DOM

В справочниках *DOM* описывается так:

DOM (от англ. Document Object Model — «объектная модель документа») — это независящий от платформы и языка программный интерфейс (API), который позволяет программам и скриптам получить доступ к содержимому HTML-, XHTML- и XML-документов, а также изменять содержимое, структуру и оформление таких документов.

DOM — это универсальный механизм, разные языки программирования его используют, чтобы получить доступ к разным типам документов.

Мы же будем рассматривать *DOM* как представление веб-страницы, которое доступно для изменения через JavaScript.

Используя *DOM*, JavaScript может изменять атрибуты элементов, создавать новые или удалять существующие элементы с веб-страницы и т. п.

Важно, что все изменения в *DOM* немедленно появляются на веб-странице. Фактически, через JavaScript-код вы изменяете содержимое веб-страницы «на ходу».

Но прежде чем выяснить, как же управлять веб-страницей при помощи JavaScript, разберём, кто и как формирует *DOM*.

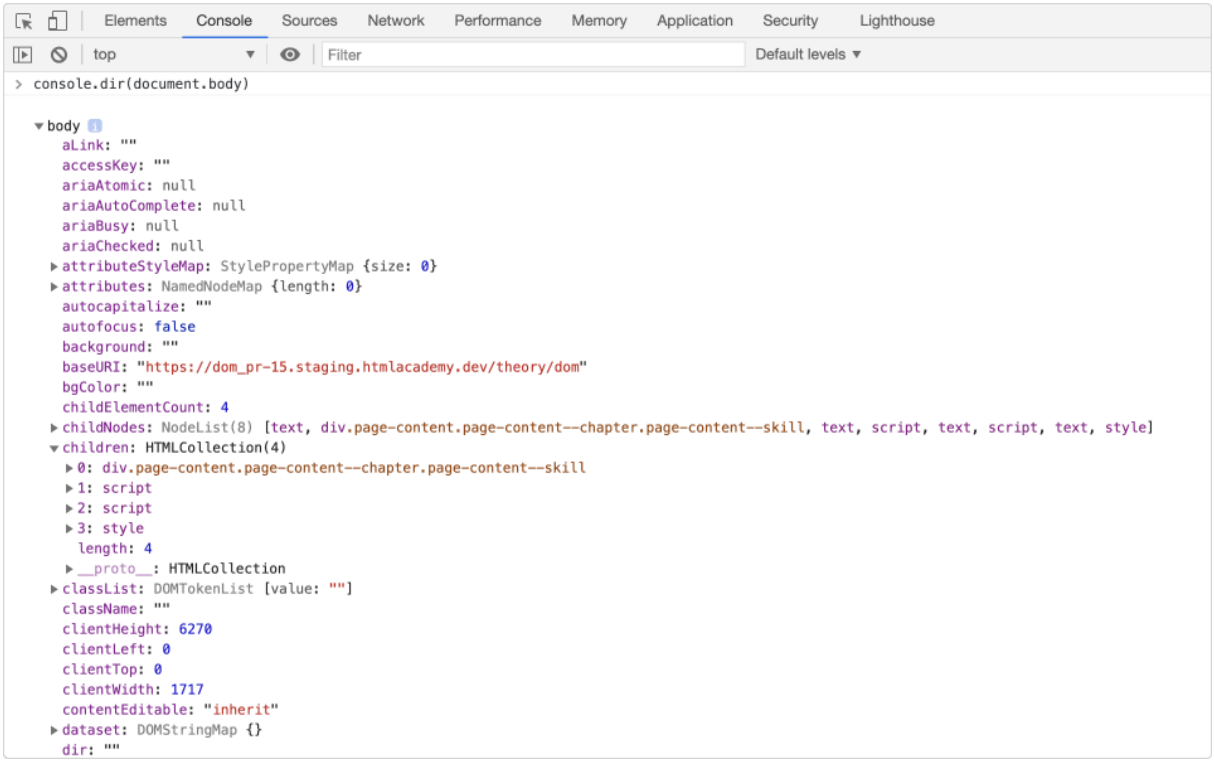
DOM создаёт браузер при загрузке веб-страницы. Он не только разбирает и выводит на экран *HTML*, но и создаёт набор объектов, которые представляют эту разметку. Созданные объекты сохраняются в виде узлов *Объектной модели документа*.

DOM-узлы — это обычные *JavaScript*-объекты. Для наследования они используют классы, основанные на прототипах.

Чтобы проверить свойства DOM-узла, можно использовать команду **console.dir()**. Эта команда выводит в консоль JSON-представление переданного объекта, что удобно для анализа его свойств. Подробно команда **console.dir()** описывается в справочнике по API браузера [1].

Например, вот как выглядит тело документа — тег <body> в виде объекта. В консоли мы ввели

```
console.dir(document.body).
```



Тег `<body>` в виде объекта

Каждый *DOM*-узел принадлежит определённому встроенному классу. Есть базовый класс для всех узлов — класс *Node*. От него наследуются другие типы узлов. Есть узлы-элементы (класс *Element*), текстовые узлы (класс *Text*), узлы-атрибуты (класс *Attr*), узлы-комментарии (класс *Comment*) и другие.

Схема взаимных связей классов DOM-узлов изображена на рис.1 ниже:

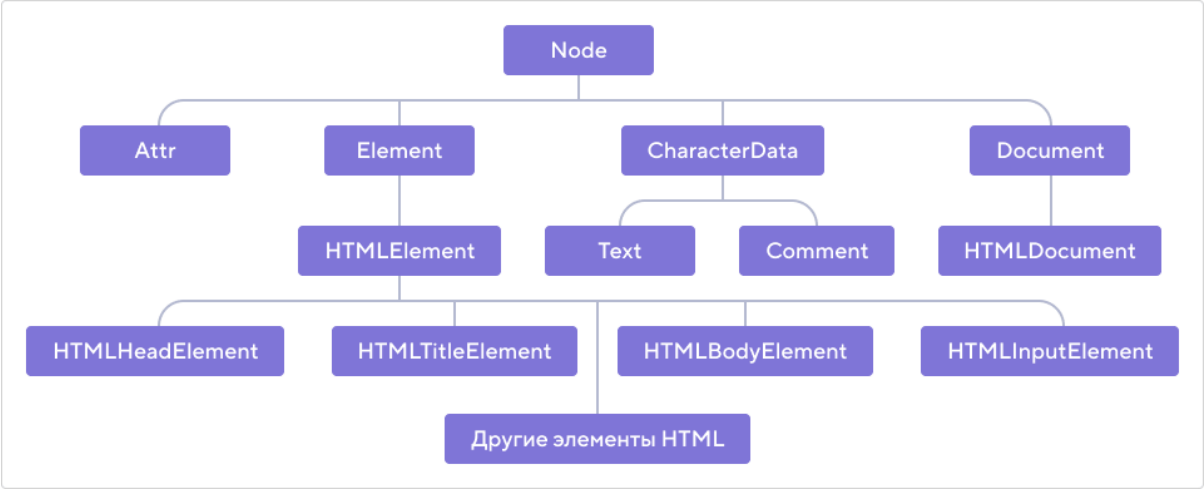


Рисунок 1: – Схема взаимных связей классов DOM-узлов

Теги образуют узлы-элементы (*Element*), текст внутри тегов формирует текстовые узлы (*Text*), атрибуты тегов — узлы-атрибуты (*Attr*), комментарии в разметке преобразуются в узлы-комментарии (*Comment*).

Пробелы ␣ и переводы строки ↵ (в *JavaScript* он обозначается как \n) — это полноправные символы, как буквы и цифры. Они также образуют текстовые узлы и становятся частью дерева *DOM*.

Рассмотрим на примере. У нас есть страница:

```
<!DOCTYPE HTML>
<html lang="ru">
<head>
  <title>Вся правда о DOM</title>
</head>
<body>
  <!-- Содержимое страницы -->
  <h1 class="center">Document Object Model</h1>
  <p>Структура DOM похода на перевёрнутое дерево: корень расположен наверху, а листья - внизу.</p>
  <p>Корень DOM – объект document.</p>
  <p>Чаще всего DOM-узлы – это теги в разметке страницы. Но есть и текстовые узлы, тестовое содержимое тегов.</p>
  <p>Пробелы и переводы строки – это полноправные символы, как буквы и цифры. Они также образуют текстовые узлы и становятся частью дерева DOM.</p>
</body>
</html>
```

Структура *DOM* для этой страницы показана на рис.2.

В теге <head> есть перевод строки и два пробела перед <title>. Они образовали текстовый узел #text. Аналогичная ситуация с тегами <body> и <p>, перед каждым <p> есть перевод строки и два пробела, которые превратились в узлы #text. Комментарий отобразился как узел #comment. Атрибуты lang="ru" и class="center" также сформировали отдельные узлы.

Тип DOM-узла определяет набор свойств и методов, которые ему доступны.

Подробности обо всех типах DOM-узлов можно найти в спецификации [2]. Для описания классов DOM в ней используется специальный язык Interface description language (IDL).

Всё, что есть в HTML: теги, атрибуты тегов, текст, комментарии — есть в DOM. Более того, в DOM содержится масса полезных для разработчика вещей, которые упрощают работу с веб-страницей.

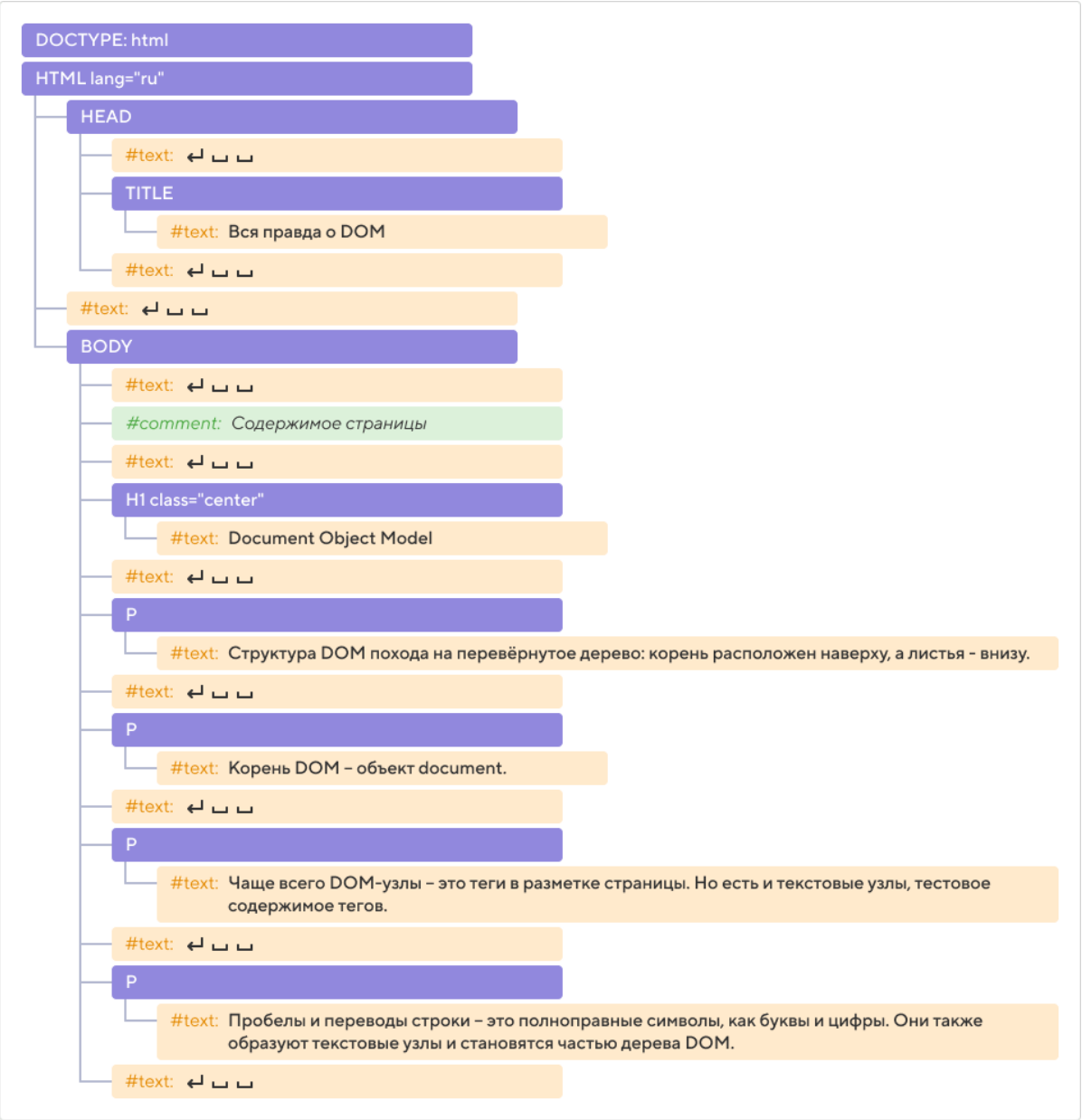


Рисунок 2: – Document Object Model для страницы

Объект *document*

На верхнем уровне DOM всегда находится объект document. Этот объект используется в JavaScript, чтобы получить доступ ко всей модели DOM.

Часто в учебниках по JavaScript можно увидеть упрощённую схему DOM — это дерево, которое состоит только из узлов-элементов. Этот вариант, во-первых, проще для восприятия, а во-вторых, JavaScript взаимодействует, как правило, именно с тегами. Чтобы получить и поменять содержимое тегов или атрибутов, используют методы и свойства узлов-элементов.

Перерисуем DOM для нашего примера на упрощённый вариант и добавим коревой объект document (рис.3):

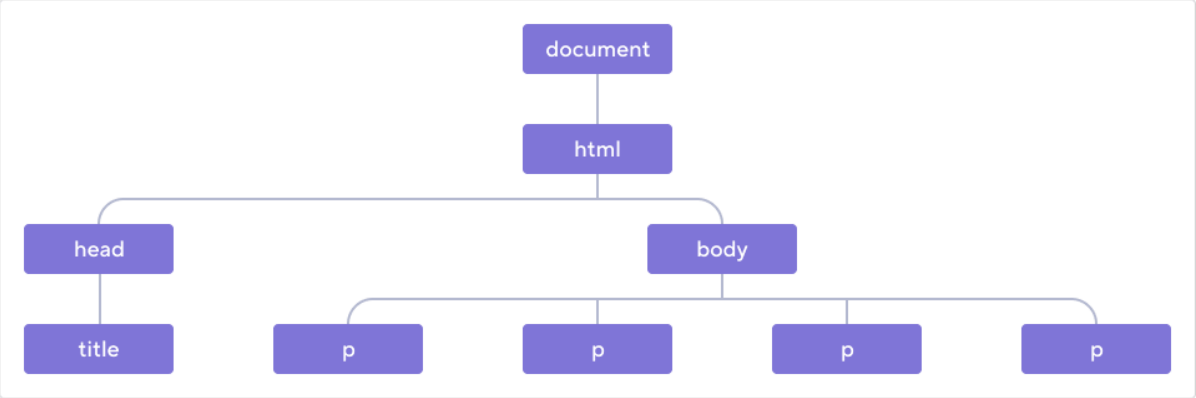


Рисунок 3: – Упрощённая модель *DOM* с корневым элементом `document`

Отметим важные моменты для *DOM*:

- *DOM* — это древовидная структура, в которой узлы связаны отношениями «родительский-дочерний»;
- корень в *DOM*-дереве может быть только один — объект document;
- у каждого *DOM*-узла может быть только один родитель;
- *DOM*-узлы — это *JavaScript*-объекты разных типов (тип узла определяет набор свойств и методов, которые ему доступны).

Инструменты для работы с DOM:

- Инструменты разработчика браузера.

  В *Google Chrome:* Дополнительные инструменты > Инструменты разработчика (**Ctrl+Shift+I** на Windows, **Cmd+Opt+I** на macOS). Могут пригодится вкладки *Elements*, *Sources*, *Console*.

  В *Mozilla FireFox*: Веб-разработка > Инструменты разработчика (**Ctrl+Shift+I** на Windows, **Cmd+Opt+I** на macOS). Полезные вкладки: *Инспектор*, *Консоль*, *Отладчик*.

- Live DOM Viewer— инструмент [3 http://software.hixie.ch/utilities/js/live-dom-viewer/], куда можно закинуть разметку и посмотреть сформированный *DOM*. Дополнительный его плюс, сразу видны автоисправления, которые добавляет браузер.

- Автоисправление

  Если браузер сталкивается с неправильно написанным HTML-кодом, он автоматически корректирует его при построении DOM.

  В начале документа всегда должен быть тег <html>. Даже если его нет в документе — браузер его создаст, и он будет в дереве DOM. То же самое касается и тега <body>.

  О других исправлениях, которые автоматически добавляются в DOM, можно узнать в спецификации [https://dom.spec.whatwg.org/].

2. Поиск элементов в DOM

Как мы уже выяснили, JavaScript получает доступ к элементам и их содержимому через DOM и может выполнять любые манипуляции над ними. Но перед тем, как мы будем изменять элементы, надо их найти. Работа с DOM-деревом всегда начинается с поиска. Сначала надо найти какой-то элемент или часть документа, а потом выполнить действия над ними, изменить свойства, содержимое, обработать события и другие.

Есть несколько способов найти нужные элементы.

Начнём с более старых методов, которые сейчас практически не используются. Но вы ещё можете их встретить в коде.

Методы getElementById, getElementsBy*

getElementById – возвращает ссылку на объект типа Element с указанным id или null. Если такой элемент отсутствует, метод вернёт null. Параметр id чувствителен к регистру. Метод можно вызвать только у объекта document.

*document.getElementsBy** – семейство методов. Эти методы используются для поиска элементов по тегу, классу и так далее и возвращают коллекцию элементов. В отличие от getElementById, методы можно вызывать у любого DOM-узла, а не только у объекта document.

Методы querySelector, querySelectorAll

Следующая группа методов: *querySelector* и *querySelectorAll*. Эти методы сочетают в себе возможности методов семейства *getElementsBy**.

***querySelector* —** метод, который возвращает первый элемент (объект класса Element) документа, который соответствует указанному селектору или группе CSS селекторов. Если совпадений не найдено, возвращает значение null.

```
document.querySelector('селектор');
```

Эта инструкция состоит из двух частей. Первая часть — элемент, внутри которого будет искать JavaScript. Словом *document* обозначается веб-страница, к которой скрипт подключён. Неважно, как называется файл на самом деле, в JavaScript это всегда «документ». Он является элементом-родителем для любого другого элемента на странице.

Вторая часть инструкции — это то, что нужно сделать. Её называют методом.

```
// Поиск элемента по тегу
var list = document.querySelector('ul');
```

```
// Поиск последнего элемента из списка
var lastProduct = document.querySelector('li:last-child');

// Поиск элемента по классу
var price = document.querySelector('.price');

// Поиск третьего элемента из списка товаров
var thirdProduct = document.querySelector('.product:nth-
child(3)');

// Поиск всех элементов, подходящих по селектору
var listItems = document.querySelectorAll('.product');
```

querySelectorAll возвращает список (коллекцию) элементов. Этот список похож на массив, но им не является. Он называется **псевдомассив (**коллекция *NodeList***)** и его можно перебирать с помощью цикла for.

Добавление класса элементу страницы:

```
// Когда ищем элемент по классу, используем точку
var product = document.querySelector('.product');

// Но когда добавляем класс, точки нет!
product.classList.add('product--sale');
```

Результат работы **classList.add()** такой же, как при ручном добавлении класса в разметку:

```
<!-- Исходное состояние разметки -->
<li class="product">
  …
</li>

<!-- Состояние после вызова classList.add -->
<li class="product product--sale">
  …
</li>
```

Свойство **DOM-элемент.children** возвращает коллекцию дочерних, то есть вложенных, DOM-элементов.

Создание элемента и добавление его в DOM-дерево:

```
var list = document.querySelector('.cards');

// Создаём новый элемент
var card = document.createElement('li');

card.classList.add('card');

// После вызова этого метода новый элемент отрисуется на
странице
list.appendChild(card);
```

Вот что произойдёт с разметкой после вызова **appendChild**:

```
<!-- Исходное состояние разметки -->
<ul class="cards">
  <li class="card">Существующий элемент</li>
</ul>

<!-- Состояние после вызова appendChild -->
<ul class="cards">
  <li class="card">Существующий элемент</li>
  <li class="card">Добавленный элемент</li>
</ul>
```

Работа с текстовым содержимым элемента:

HTML

```
<p>Я — <em>текстовый элемент</em>.</p>
```

JS

```
var p = document.querySelector('p');
console.log(p.textContent);
```

Выведет: Я — текстовый элемент.

```
p.textContent = 'Теперь у меня новое содержимое.';
console.log(p.textContent);
```

Выведет: Теперь у меня новое содержимое.

В HTML содержание тега изменится

```
<p>Теперь у меня новое содержимое.</p>
```

Работа с изображениями:

Создание изображения

```
var picture = document.createElement('img');
```

Добавляем адрес картинки

```
picture.src = 'images/picture.jpg';
```

Добавляет альтернативный текст

```
picture.alt = 'Непотопляемая селфи-палка';
```

Добавляет изображение в конец родительского элемента

```
element.appendChild(picture);
```

Консоль

Консоль — инструмент разработчика, который помогает тестировать код. Если во время выполнения скрипта возникнет ошибка, в консоли появится сообщение о ней. А ещё в консоль можно выводить текстовые подсказки. Чтобы вывести сообщение в консоль, нужно использовать `console.log`:

```
console.log('Привет от JavaScript!');
```

Выведет: Привет от JavaScript!

```
console.log(document.querySelector('.page'));
```

Выведет в консоль найденный элемент

Переменная

Переменная — способ сохранить данные, дав им понятное название.

Переменную можно создать, или объявить, с помощью ключевого слова *let*. За ним следует имя переменной. После объявления в переменную нужно записать, или присвоить, какое-то значение:

```
let header = document.querySelector('header');
```

Имя переменной может быть почти любым, но не должно начинаться с цифры, а из спецсимволов разрешены только '_' и '$'. Ещё для именования переменных нельзя использовать зарезервированные слова. Имена переменных чувствительны к регистру: header, Header и HEADER — это разные переменные. Имя переменной должно описывать то, что в ней хранится.

Когда в коде встречается переменная, браузер вместо её имени подставляет присвоенное ей значение. Когда мы используем переменную, снова писать let не нужно:

```
console.log(header);
```

Ключевое слово *let* появилось в JavaScript в 2015 году, до этого для объявления переменных использовалось слово *var*.

Методы для изменения классов

Чтобы убрать у элемента класс, нужно использовать метод *classList.remove*. Он убирает с элемента тот класс, который указан в скобках:

```
элемент.classList.remove('класс');
```

Чтобы добавить элементу класс, нужно использовать метод *classList.add*:

```
элемент.classList.add('класс');
```

Метод-переключатель classList.toggle убирает у элемента указанный класс, если он есть, и добавляет, если этого класса нет:

```
элемент.classList.toggle('класс');
```

Свойство textContent

У каждого элемента имеется множество свойств: его размеры, цвет и так далее. Свойство textContent хранит в себе текстовое содержимое элемента. Свойствам можно присваивать новые значения:

```
let paragraph = document.querySelector('p');
paragraph.textContent = 'Здесь был Макс. Гав!';
```

Свойство value

У полей ввода есть особое свойство — value. Оно хранит данные, введённые в поле. Мы можем вывести их прямо на страницу:

```
let input = document.querySelector('input');
paragraph.textContent = input.value;
```

Конкатенация

Операция, когда мы «склеиваем» несколько значений, называется конкатенацией и в JavaScript выполняется с помощью знака плюс.

```
let name = 'Макс';
paragraph.textContent = 'Вас зовут ' + name + '. Хорошего дня!';
console.log(paragraph.textContent);
```

Выведет: Вас зовут Макс. Хорошего дня!

```

```

3. Знакомство с событиями

События — действия пользователя на странице (клик по кнопке, нажатие клавиши на клавиатуре и т.д.).

Добавление обработчиков событий

```
button.addEventListener('click', function () {
  // Инструкции
});
```

В примере:

- `button` — элемент, на котором мы хотим «слушать» событие.

- `addEventListener()` — функция добавления обработчика события на элемент.
- `'click'` — общепринятое название события, первый параметр функции `addEventListener`. Названия всех событий можно посмотреть здесь [2 Event reference | MDN (mozilla.org)].
- Второй параметр `addEventListener` — функция-обработчик, в ней записаны инструкции, которые выполнятся, только **когда произойдёт событие**.

  Обратите внимание, мы передаём функцию, **а не её вызов**. Если мы вызовем функцию, код из этой функции выполнится сразу и больше не сработает. А нам нужно, чтобы код выполнился асинхронно — в момент, когда произойдёт событие.

  Так добавлять обработчик неправильно

```
button.addEventListener('click', function () {
  console.log('Клик по кнопке');
}());
```

  Сообщение сразу же выведется в консоль. А такой код верный

```
button.addEventListener('click', function () {
  console.log('Клик по кнопке');
});
```

  Сообщение выведется, когда произойдёт событие клика

  В примере выше мы передаём в обработчик функцию, у которой нет своего имени, она не записана в переменную. Мы создали её там же, где передаём. Такие функции, которые создаются в момент передачи и не имеют имени, называются анонимными функциями.

Объект event

Объект `event` — параметр функции-обработчика. Он всегда передаётся браузером в эту функцию в момент наступления события. Этот объект содержит много полезных свойств и методов.

Чтобы использовать `event`, достаточно указать этот объект параметром функции-обработчика и написать инструкции. Остальное сделает JavaScript. Среди некоторых разработчиков принято называть параметр сокращённо — `evt`, во избежание ошибок.

Действия по умолчанию

Некоторые элементы страницы имеют действия по умолчанию или дефолтные действия. Например, клик по кнопке отправления формы вызывает отправку данных этой формы на сервер, а при клике по ссылке браузер переходит по этой ссылке.

Объект event содержит метод, который отменяет действие элемента по умолчанию: preventDefault().

```
link.addEventListener('click', function(evt) {
  // Отменяем действие по умолчанию
  evt.preventDefault();

  // Добавляем инструкции для события клика
  console.log('Произошёл клик');
});
```

Обработчики событий onclick и onsubmit

JavaScript следит за всем, что происходит на странице. Клик по кнопке или отправка формы — это событие. Мы можем сказать JavaScript, что сделать, когда некое событие произойдёт. Для этого используют обработчики событий. Инструкции, которые должны будут выполниться, когда событие произойдёт, располагают между фигурных скобок.

Свойство onclick означает «по клику»:

```
let button = document.querySelector('button');
button.onclick = function() {
  console.log('Кнопка нажата!');
};
```

При каждом клике по кнопке в консоли будет появляться новое сообщение “Кнопка нажата!”.

За обработку отправки формы отвечает свойство onsubmit:

```
let form = document.querySelector('form');
form.onsubmit = function() {
  console.log('Форма отправлена!');
};
```

После отправки формы в консоли появится сообщение Форма отправлена!.

Клавиатурные события

У события «нажатие на клавишу» есть специальное название — 'keydown'. Такое событие срабатывает при нажатии на любую клавишу. Обратите внимание, слушать это событие можно только на элементах, которые имеют состояние фокуса: поля ввода, кнопки, элементы с атрибутом tabindex, документ. При нажатии фокус должен находиться на соответствующем элементе.

Если мы хотим поймать нажатие какой-то конкретной клавиши, можно обратиться к свойству keyCode объекта event. Это свойство содержит код нажатой клавиши. Например, у Enter код 13, а у ESC — 27. Эти номера универсальны и одинаковы в любой раскладке. Найти код любой клавиши можно здесь [5 KeyboardEvent.keyCode - Web APIs | MDN (mozilla.org)].

```
document.addEventListener('keydown', function(evt) {
  // Проверяем, что код клавиши равен 27
  if (evt.keyCode === 27) {
    // Код отсюда выполнится только при нажатии ESC
  }
});
```

Кроме keyCode есть и другие свойства для определения нажатой клавиши. Например, key и code. Они возвращают названия клавиш, а не их номера. Эти свойства пока поддерживаются не во всех браузерах, но когда поддержка станет лучше, стоит начать использовать их вместо keyCode в соответствии с современным стандартом JavaScript.

Области видимости

У каждой функции есть область видимости — все значения, доступные для этой функции.

Область видимости ограничена функцией, поэтому снаружи нельзя получить локальные переменные и параметры функции.

Локальные переменные — переменные, у которых область видимости ограничена функцией, где они объявлены. Такая область видимости называется локальной.

Глобальные переменные — переменные, которые объявлены на уровне всей программы, их видно из любого блока кода. Область видимости, в которой они объявлены, называется глобальной.

Если внутри функции обратиться не к локальной переменной, JavaScript будет искать переменную снаружи, переходя наверх от уровня к уровню, пока не найдёт переменную. Если переменной не будет ни внутри функции ни снаружи, будет ошибка.

Так как функция может использовать переменные, объявленные снаружи, их можно переопределять.

```
var food = 'макароны';

var eatDinner = function () {
  console.log('Поел ' + food);
};

eatDinner();
// Выведет: Поел макароны

// Переопределяем переменную food
food = 'сельдерей';

eatDinner();
// Выведет: Поел сельдерей
```

Переопределять снаружи переменные, которые использует функция — не лучшая практика. Это может приводить к неожиданным последствиям и ошибкам в коде. Использовать это нужно осторожно.

Области видимости создаются только функциями. Поэтому, если переменная была создана в другой конструкции, например, в цикле, она будет доступна для чтения из функции.

Замыкания

Замыкание — функция, которая помнит о своём окружении. Это функция + все значения вне локальной области видимости, которые она использует.

Благодаря замыканиям мы можем зафиксировать какое-то значение в функции, а использовать саму функцию позже.

```
var collectContainer = function (food) {
  return function () {
    console.log('Поел ' + food);
  };
};

var schoolkid = collectContainer('макароны');
schoolkid();
// Выведет: Поел макароны
```

Замыкания и асинхронность

Некоторые функции выполняются асинхронно, поэтому в момент выполнения кода значение переменной может уже измениться. Чтобы избавиться от этой проблемы, нужно создать отдельную область видимости. Так все переменные будут под контролем и замыкания не позволят потерять необходимые значения.

```
var thumbnails = document.querySelectorAll('.gallery__photo-
preview');
var fullPhoto = document.querySelector('.full-photo');

var addThumbnailClickHandler = function (thumbnail, photo) {
  thumbnail.addEventListener('click', function () {
    fullPhoto.src = photo;
  });
};

for (var i = 0; i < thumbnails.length; i++) {
  addThumbnailClickHandler(thumbnails[i], photos[i]);
}
```

4. Методика работы с DOM

4.1 «Оживление» страницы за счёт изменения в DOM

Как правило, на динамических сайтах работает два сценария:

- изменение свойств у элементов, которые уже есть на странице,

- добавление новых элементов, которые создаются по заранее известному шаблону.

Обычно оба сценария срабатывают, когда пользователь совершил определённые действия, нажал кнопку, отметил чекбокс и так далее. Иногда действие выполняется при наступлении события (загрузка страницы, окончание загрузки данных и подобные).

Первым делом проанализируем типовые ситуации на сайтах, которые подходят под описанные выше сценарии. А после анализа попробуем составить схему взаимодействия с DOM и выделим основные шаги для реализации нужного функционала.

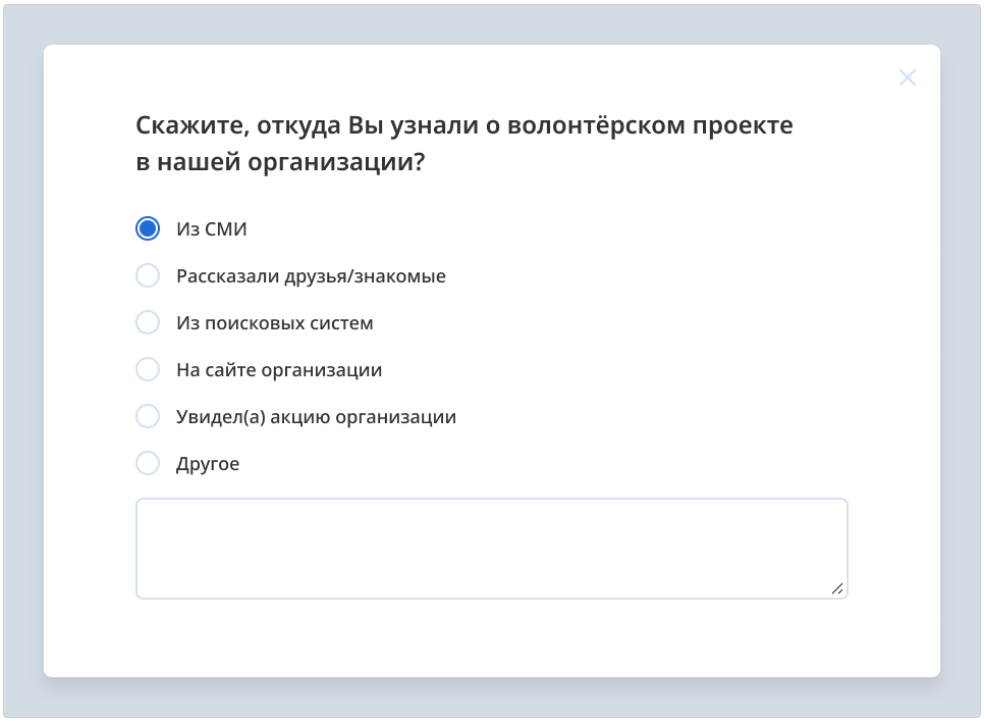
В методике рассмотрим основные схемы «оживления» страницы за счёт JavaScript-кода:

- при изменении свойств элементов страницы,
- при добавлении новых элементов в DOM.

Изменение свойств элементов страницы

Вот несколько типовых ситуаций.

Первая: пользователь заполняет форму опросника. В пункте *«Скажите, откуда Вы узнали о волонтёрском проекте в нашей организации?»* ему предлагается выбрать только один из вариантов ответа, также есть вариант *«Другое»*, который делает доступным для изменения текстовое поле. По умолчанию текстовое поле заблокировано для ввода.



Форма с вариантом Другое

Получается, что при изменении отметки в группе радиокнопок нужно выяснить, какая из них отмечена. Если это вариант «Другое», снимать

атрибут disabled с текстового поля, иначе добавлять этот атрибут. Нам понадобится обработчик события change у формы. На ней через делегирование будет отлавливать изменение состояния радиокнопок. Но прежде надо найти элементы. Ищем форму и текстовое поле, для которого будем убирать/добавлять атрибут disabled.

Разметка для опросника:

```
<form class="feedback" action="#" method="POST">
  <fieldset>
    <legend>Скажите, откуда Вы узнали о волонтёрском проекте в нашей организации?</legend>
    <label>
      <input type="radio" name="source" id="news" value="news" checked>
      Из СМИ
    </label>
    <label>
      <input type="radio" name="source" id="friends" value="friends">
      Рассказали друзья/знакомые
    </label>
    <label>
      <input type="radio" name="source" id="search" value="search">
      Из поисковых систем
    </label>
    <label>
      <input type="radio" name="source" id="site" value="site">
      На сайте организации
    </label>
    <label>
      <input type="radio" name="source" id="share" value="share">
      Увидел(а) акцию организации
    </label>
    <label>
      <input type="radio" name="source" id="other" value="other">
      Другое
    </label>
```

```
    <textarea name="other-answer" id="other-answer" disabled>
    </textarea>
  </fieldset>
</form>
```

Код с решением:

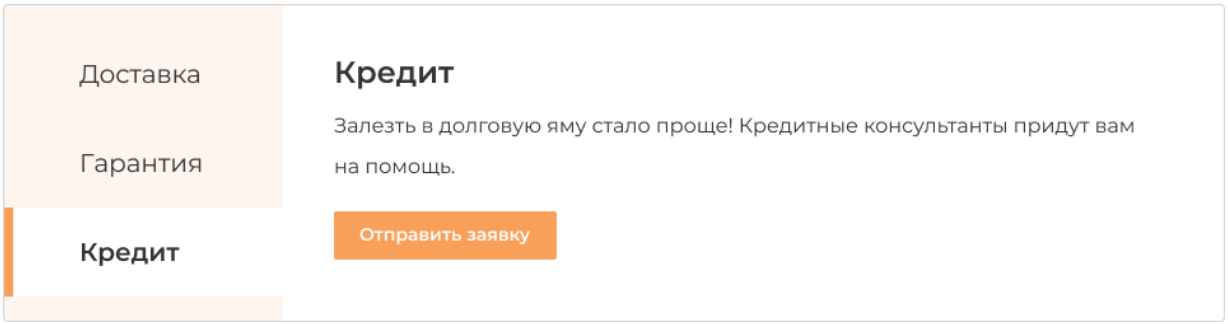
```
// нашли форму
const feedbackForm = document.querySelector('.feedback');
// нашли поле для ввода текста
const textField = document.querySelector('#other-answer');

const formChangeHandler = (evt) => {
  // нас интересует только изменение состояний радиокнопок
  const item = evt.target.closest('input[type="radio"]');

  if (!item) {
    return;
  }

  // если изменяется состояние радиокнопки Другое и она
становится отмеченной, убираем атрибут disabled
  textField.toggleAttribute('disabled', item.id !== 'other' ||
!item.checked);
}

// подписались на событие изменение формы (через него будем
отлавливать изменение радиокнопок)
feedbackForm.addEventListener('change', formChangeHandler);
```

Другая ситуация

В разделе добавлена внутренняя навигация — список ссылок и закладок. За счёт неё можно сэкономить место на странице. Переключение между закладками выполняет *JavaScript*-код. Активная ссылка может быть выделена с помощью специального CSS-класса, например, `item-active`. При клике на ссылку будем добавлять ей этот класс, при этом убирать его с других ссылок. Скрыть закладки также можно за счёт CSS. Допустим, по умолчанию закладки не показываем, то есть `display: none;`, а для активной устанавливаем `display: block;`. Активную закладку также выделяем специальным классом — `page-active`.

Внутренняя навигация на странице

Активна закладка Гарантия.



Переключение на другую вкладку

А теперь переключились на закладку Кредит.

Составим разметку для блока. Ссылку и соответствующую ей страницу можно связать через атрибуты href и id.

Вот, что получилось:

```
<div class="services">
  <ul class="services-navigation">
    <li class="services-navigation-item">
      <a class="services-navigation-link" href="#page-1">Доставка</a>
    </li>
    <li class="services-navigation-item">
      <a class="services-navigation-link" href="#page-2">Гарантия</a>
    </li>
    <li class="services-navigation-item">
      <a class="services-navigation-link item-active" href="#page-
3">Кредит</a>
    </li>
  </ul>
  <div class="services-content">
    <section class="page" id="page-1">
      <h3>Доставка</h3>
      <p>Мы с удовольствием доставим ваш товар прямо к вашему подъезду
совершенно бесплатно! Ведь мы неплохо заработаем, поднимая его на ваш
на этаж!</p>
    </section>
    <section class="page" id="page-2">
      <h3>Гарантия</h3>
      <p>Если купленный у нас товар поломается или заискрит, а также в
случае пожара, спровоцированного его возгоранием, вы всегда можете
быть уверены в нашей гарантии. Мы обменяем сгоревший товар на новый.
Дом уж восстановите как нибудь сами.</p>
```

```
    </section>
    <section class="page page-active" id="page-3">
      <h3>Кредит</h3>
      <p>Залезть в долговую яму стало проще! Кредитные консультанты придут вам на помощь.</p>
      <a href="#">Отправить заявку</a>
    </section>
  </div>
</div>
```

Опишем стили:

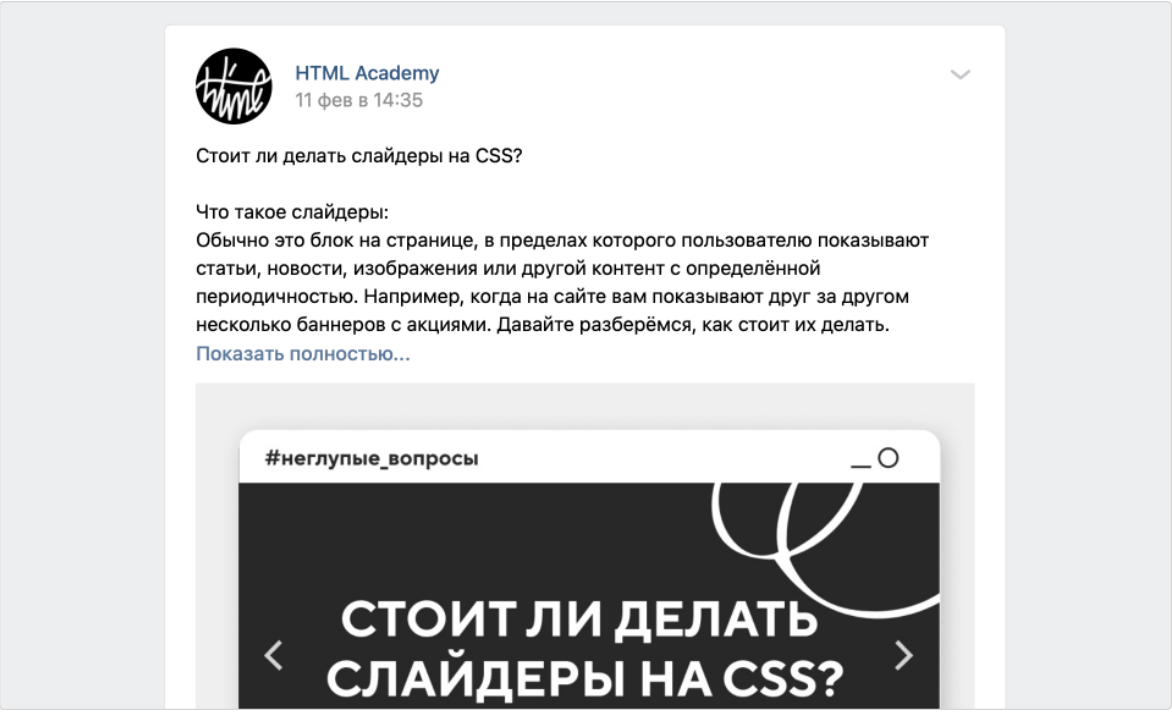
```
...
.services-navigation-link {
  padding: 10px 15px;
  color: inherit;
}

/* активная ссылка */
.services-navigation-link.item-active {
  text-decoration: none;
}

/* по умолчанию страница скрыта */
.services-content section {
  display: none;
}

/* активная страница */
.services-content .page-active {
  display: block;
}
```

В JavaScript-коде найдём элемент с навигацией. Также найдём элементы с ссылками и элементы со страницами. Для элемента с навигацией добавим обработчик события click (снова используем делегирование, чтобы не создавать много обработчиков событий). А дальше всё просто: при клике на ссылку добавляем ей класс item-active и удаляем его у других ссылок. С помощью метода getAttribute узнаем идентификатор страницы, которая станет активной. По этому идентификатору добавляем класс page-active для будущей активной страницы и удаляем его у других страниц.

```
// Поиск элементов
// Навигация
const nav = document.querySelector('.services-navigation');
// отдельные ссылки в навигации
const navItems = document.querySelectorAll('.services-navigation-link');
// элементы со страницами
const pages = document.querySelectorAll('.page');

const setActivePage = (anchor) => {
  // делаем активным пункт меню (переключаем стили), ориентир - значение атрибута href
  navItems.forEach((navItem) => {
```

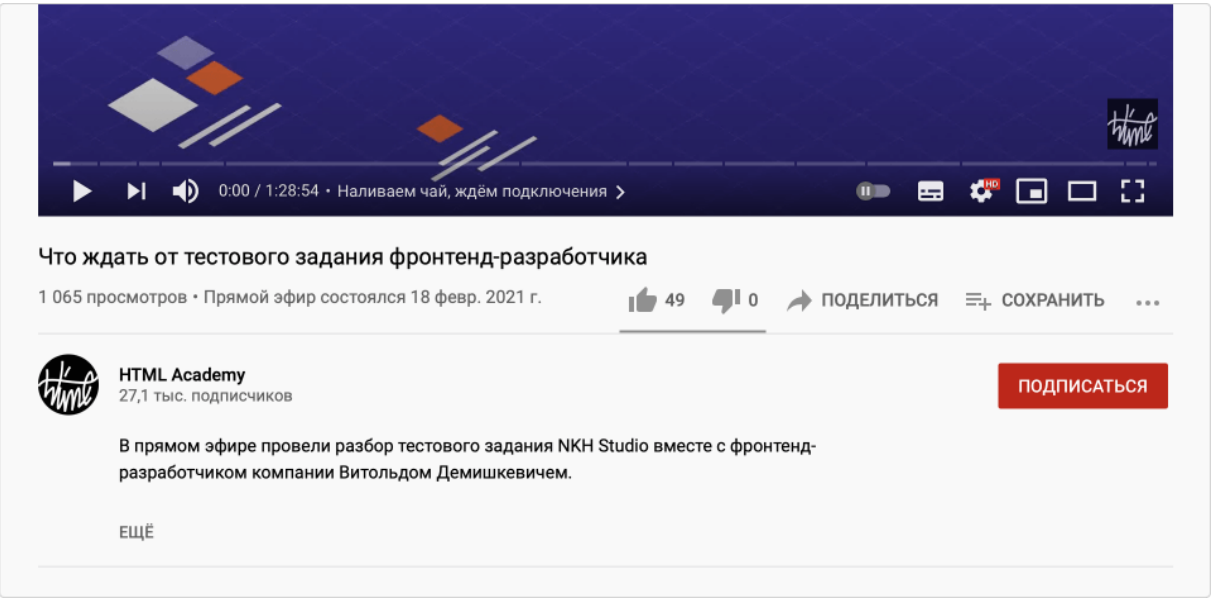
```
    navItem.classList.toggle('item-active', anchor ===
navItem.getAttribute('href'));
  });

  // показываем страницу (переключаем стили), на которую ссылается
пункт меню
  pages.forEach((page) => {
    page.classList.toggle('page-active', anchor === ('#' + page.id));
  });
}

const navClickHandler = (evt) => {
  // используем делегирование и обрабатываем Click только на ссылках в
навигации
  const link = evt.target.closest('a');

  if (!link) {
    return;
  }

  // получаем значение якоря, по нему будем открывать соответствующую
страницу
  const anchor = link.getAttribute('href');

  // переключение страницы по значению якоря
  setActivePage(anchor);
}

const init = () => {
  // подписываемся на событие Click в навигации
  nav.addEventListener('click', navClickHandler);
}

// начальная инициализация элементов страницы
init();
```

3. **Ещё одна типовая ситуация**: карточки с объёмным текстовым содержимым и ссылкой *Показать полностью...*.

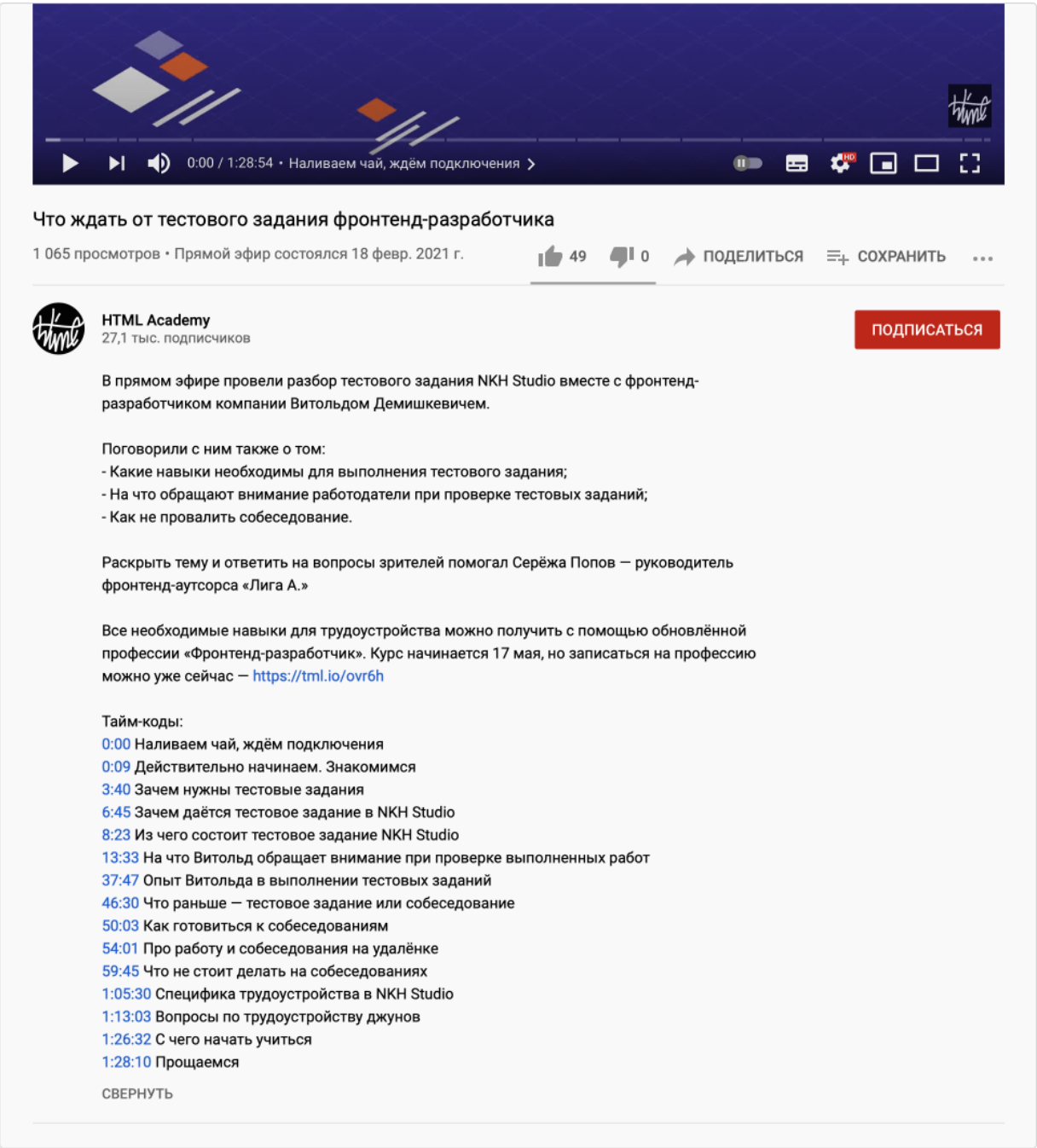


Элемент Показать полностью... на странице

Или так: подробную информацию можно показать, а затем снова свернуть.



Элемент Ещё



Элемент Свернуть

Реализуем второй вариант, то есть будем скрывать и показывать полное описание по клику на ссылку *Ещё*/*Свернуть*. Скрыть описание можно за счёт стилей — установить максимальную высоту блока, а всё, что не поместилось, обрезать при помощи `overflow: hidden;`. Опишем CSS-класс `description-short` с соответствующими свойствами. При клике на ссылку будет переключаться этот класс и дополнительно меняться текст ссылки, в зависимости от того, какое сейчас описание — полное или сокращённое.

Итак, разметка:

```
<div class="content">
  <div class="description description-short">
```

```
    <p>В прямом эфире провели разбор тестового задания NKH Studio
вместе с фронтенд-разработчиком компании Витольдом Демишкевичем.</p>
  </div>
  <a class="more" href=""#>Ещё</a>
</div>
```

Стили:

```
.content {
  padding: 20px 0;
  width: 570px;
}

/* класс для свёрнутого описания */
.description-short {
  overflow: hidden;
  max-height: 60px;
}
```

Код:

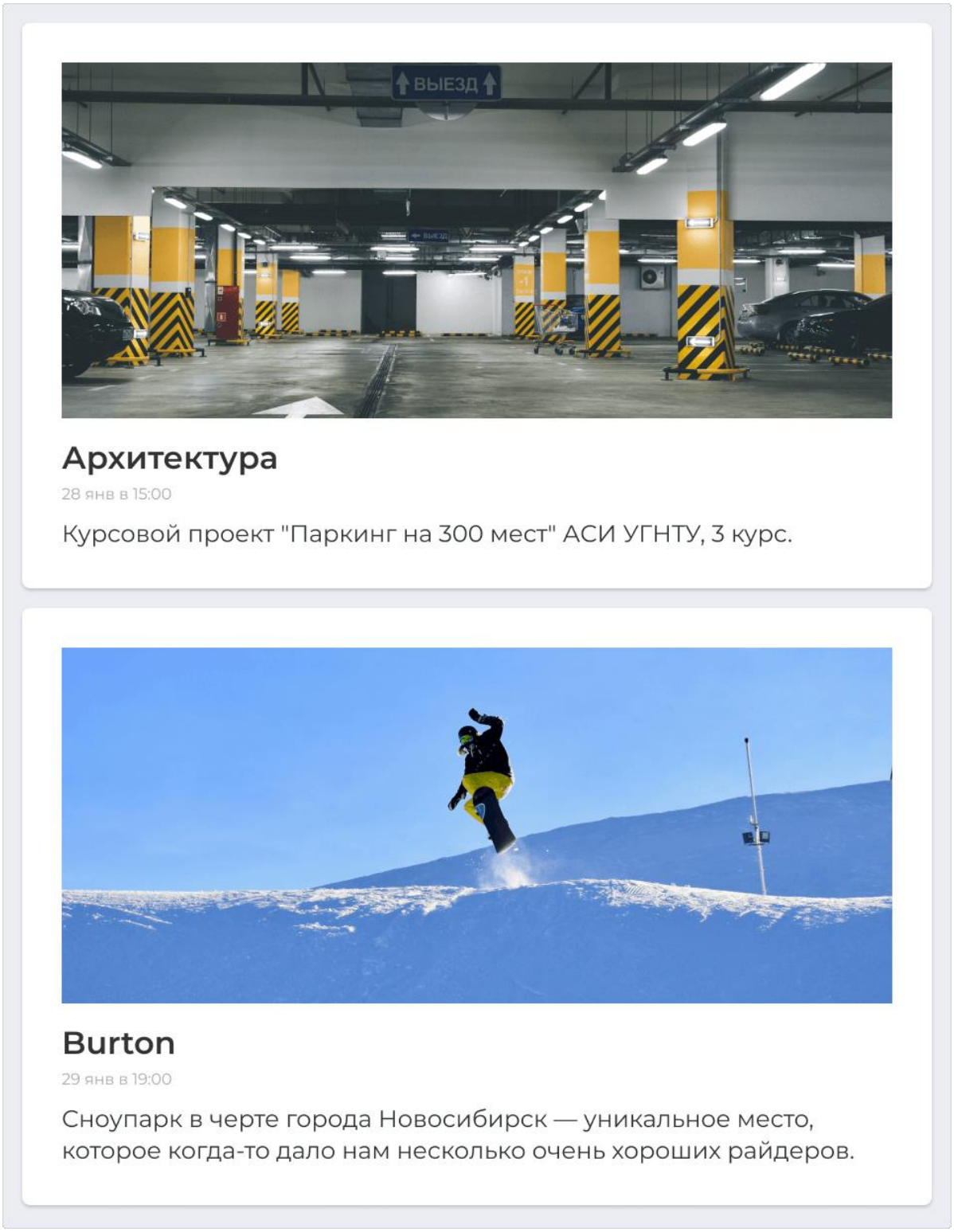
```
const content = document.querySelector('.content');

const moreLinkClickHandler = (evt) => {
  // отлавливаем событие только на ссылке с классом more
  const element = evt.target.closest('a.more');

  if (!element) {
    return;
  }
  // отменяем для ссылки действие по умолчанию
  evt.preventDefault();

  const descriptionElement = document.querySelector('.description');

  if (descriptionElement) {
    // переключаем класс
    descriptionElement.classList.toggle('description-short');
    // меняем текст ссылки
    element.textContent =
descriptionElement.classList.contains('description-short') ? 'Ещё' :
'Свернуть';
  }
};

content.addEventListener('click', moreLinkClickHandler);
```

Если не вдаваться в подробности, то схема работы во всех описанных случаях одинаковая:

- сначала — поиск элемента,
- затем изменение свойств у элемента (это может быть добавление/удаление класса, добавление/удаление атрибута, изменение содержимого элемента и так далее)

4.2 Добавление новых элементов в DOM

Добавиться может группа однотипных элементов, например, новости в ВКонтакте.



Новости в ВКонтакте

Разберём эту ситуацию подробнее. Новые посты появляются ниже уже существующих (бесконечная загрузка и вставка после). Как это работает в упрощённом варианте.

У нас есть контейнер с записями . В нём содержатся блоки с отдельными новостями <section class="post">. Опишем разметку блоков:

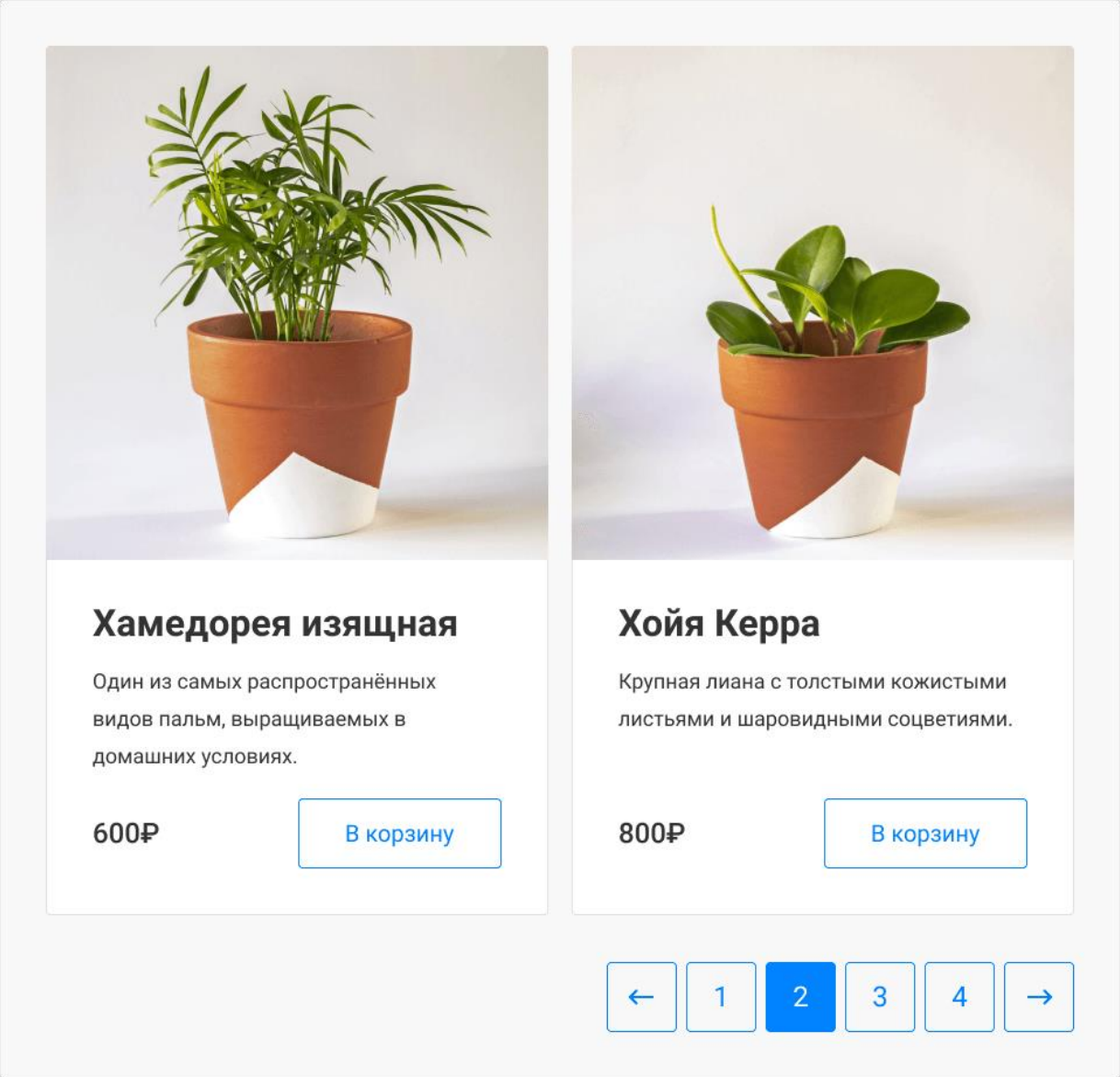
```
<div class="posts">
  <section class="post">
    <header class="post-header">
      <h2>Архитектура</h2>
      <img src="img/pic-21.jpg" width="200" height="150" alt="">
      <time datetime="2021-01-28 15:00">28 янв в 15:00</time>
    </header>
    <div class="post-content">
      <p>Курсовой проект "Паркинг на 300 мест" АСИ УГНТУ, 3 курс</p>
    </div>
  </section>
  <section class="post">
    <header class="post-header">
      <h2>Burton</h2>
      <img src="img/pic-25.jpg" width="200" height="150" alt="">
      <time datetime="2021-01-29 19:00">29 янв в 19:00</time>
    </header>
    <div class="post-content">
      <p>Сноупарк в черте города Новосибирск - уникальное место,
которое когда-то дало нам несколько очень хороших райдеров.</p>
    </div>
  </section>
</div>
```

Сначала с помощью шаблонных строки и интерполяции сформируем HTML для элементов. А затем для добавления их в DOM используем метод insertAdjacentHTML.

```
const posts = [
  {
    title: '"Мастера"',
    date: '2021-01-29 20:00',
    img: 'img/pic-21.jpg',
    text: 'Castella Shop - это единственная в Санкт-Петербурге
пекарня, которая специализируется на приготвлении кастеллы -
нежнейшего хлопкового японского бисквита',
  },
  {
    title: 'Секрет Фирмы',
    date: '2021-01-29 21:00',
    img: 'img/pic-25.jpg',
    text: 'Одного хештега в Twitter главы Tesla и SpaceX оказалось
достаточно, чтобы инвесторы кинулись скупать криптовалюту',
  }
];

// находим контейнер с постами
const postsContainer = document.querySelector('.posts');
```

```
// формируем элементы с новыми постами
// используем шаблонные строки и интерполяцию
// склеиваем всё в одну строку
const postsString = posts.map(({title, date, text}) => `
  <section class="post">
    <header class="post-header">
      <h2>${title}</h2>
      <img src="${img}" width="200" height="150" alt="">
      <time datetime="${date}">${date}</time>
    </header>
    <div class="post-content">
      <p>${text}</p>
    </div>
  </section>
  `).join('');

// добавляем посты в контейнер
postsContainer.insertAdjacentHTML('beforeend', postsString);
```

Всем знакомая пагинация (или пейджинация) — это тоже создание новых элементов, которые замещают существующие.



Пагинация на странице

Рассмотрим этот кейс на примере интернет-магазина по продаже цветочных растений. Разметка отдельной страницы с карточками может выглядеть так:

```
<div class="cards">
  <section class="card">
    <h2>Хамедорея изящная</h2>
    <img src="img/pic-34.jpg" width="100" height="100" alt="">
    <p>Один из самых распространённых видов пальм, выращиваемых в
домашних условиях.</p>
    <p>600 руб</p>
    <a href="#">В корзину</a>
  </section>
  <section class="card">
    <h2>Хойя Керра</h2>
    <img src="img/pic-37.jpg" width="100" height="100" alt="">
    <p>Крупная лиана с толстыми кожистыми листьями и шаровидными
соцветиями.</p>
    <p>800 руб</p>
```

```
    <a href="#">В корзину</a>
  </section>
</div>
```

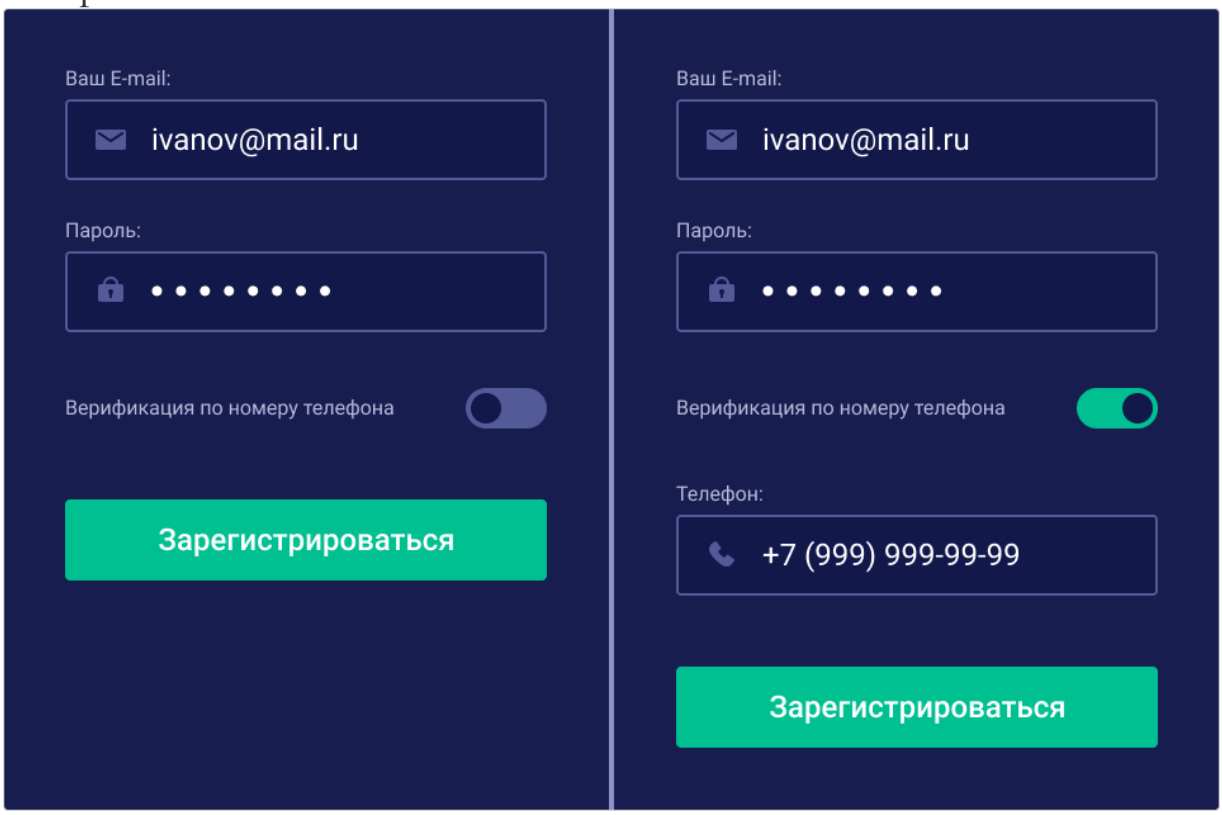
При переключении со страницы на страницу содержимое элемента очищается и взамен добавляются данные для выбранной страницы.

Так же как и в предыдущем случае, для создания элементов используем шаблонные строки и интерполяцию и метод insertAdjacentHTML для добавления в DOM. Но предварительно будем очищать родительский элемент от содержимого.

Напишем код, который реализует переключение между страницами.

```
const cards = [
  {
    title: 'Хамедорея изящная',
    img: 'img/pic-34.jpg',
    price: '600',
    description: 'Один из самых распространённых видов пальм,
выращиваемых в домашних условиях.',
  },
  {
    title: 'Хойя Керра',
    img: 'img/pic-37.jpg',
    price: '800',
    description: 'Крупная лиана с толстыми кожистыми листьями и
шаровидными соцветиями.',
  },
  ...
];

// находим родительский элемент с карточками
const cardsContainer = document.querySelector('.cards');

// очищаем содержимое <div class="cards">
cardsContainer.innerHTML = '';

// формируем HTML с новыми карточками
// используем шаблонные строки и интерполяцию
const cardsString = cards.map(({title, price, description}) => `
  <section class="card">
    <h2>${title}</h2>
    <img src="${img}" width="100" height="100" alt="">
    <p>${description}</p>
    <p>${price}</p>
    <a href="#">В корзину</a>
  </section>
  `).join('');

// заполняем элемент <div class="cards"> новыми данными
cardsContainer.insertAdjacentHTML('beforeend', cardsString);
```

Добавляться могут одиночные или разнотипные элементы. К примеру, в форме регистрации пользователь может добавить верификацию по номеру телефона.



Дополнительная верификация по номеру телефону

Поле с телефоном добавляем по нажатию на переключатель «Верификация по телефону».

```
<form class="registration" action="#" method="POST">
  <label>
    Ваш E-mail:
    <input type="email" name="email" placeholder="ivanov@mail.ru">
  </label>
  <label>
    Пароль:
    <input type="password" name="password" placeholder="password">
  </label>
  <label>
    Валидация по номеру телефона
    <input type="checkbox" class="more">
  </label>
  <div class="verification-container">
  </div>
  <button type="submit">Зарегистрироваться</button>
</form>
// находим родительский элемент для новых полей
const verificationContainer = document.querySelector('.verification-
container');

// переключатель дополнительных полей
const switchButton = document.querySelector('.more');
```

```
switchButton.addEventListener('click', (evt) => {
  // предварительно почистим родительский элемент
  verificationContainer.innerHTML = '';

  if (switchButton.checked) {
    // используем шаблонные строки
    const fieldsString = `
    <label>
      Телефон:
      <input type="phone" name="phone" placeholder="+7(999)999-99-99">
    </label>`;

    // добавляем поля в родительский элемент
    verificationContainer.insertAdjacentHTML('beforeend',
fieldsString);
  }
});
```

После анализа описанных кейсов мы можем выделить следующие шаги:

- Ищем нужный нам DOM-элемент на странице. Этот элемент будет родительским, целевым элементом или точкой отсчёта.
- Чистим прежнее содержимое найденного целевого элемента innerHTML = ''. Этот пункт выполняем, если требуется заменить прежнее содержимое элемента.
- Создаём новый (ые) элемент (ы). Используем методы: document.createElement, тег <template>, createContextualFragment или HTML-строки.
- Устанавливаем значение атрибутов, заполняем содержимое, если требуется. Используем методы: setAttribute, textContent, innerHTML и другие.
- Добавляем созданный (е) элемент (ы) в DOM. Используем элемент, который нашли в первом пункте как родительский, целевой элемент или точку отсчёта, и для этого элемента вызываем методы appendChild, append, insertAdjacentElement, insertBefore или insertAdjacentHTML.

Выбор методов зависит от того, какая браузерная поддержка нужна, добавляется один или несколько элементов, а также есть ли пользовательский ввод.

Вспомогательные материалы

1. Справочник по API браузера URL: https://developer.chrome.com/docs/devtools/console/api/

2. Стандарт DOM. URL: https://dom.spec.whatwg.org/#nodes .

3. Live DOM Viewer http://software.hixie.ch/utilities/js/live-dom-viewer/

Работа с DOM

- Что такое DOM-дерево?
- Живые и неживые коллекции в JavaScript
- Зачем нужен метод preventDefault

Функции и классы

- Всё, что вы хотели знать об областях видимости в JavaScript (но боялись спросить)
- Замыкания в JavaScript
- Введение в колбэк-функции в JavaScript
- Ключевое слово this в JavaScript — учимся определять контекст на практике
- Привязка контекста функции
- ES6: классы изнутри