Московский государственный технический университет
имени Н. Э. Баумана

В. И. Кузнецов

Работа с AJAX и DOM с использованием языка JavaScript

Методические указания к выполнению
лабораторной работы №4 по дисциплине
«Языки интернет программирования»

Редакция от 1.09.2025

Факультет «Информатика и системы управления»
Кафедра «Компьютерные системы и сети»

Кузнецов В.И.

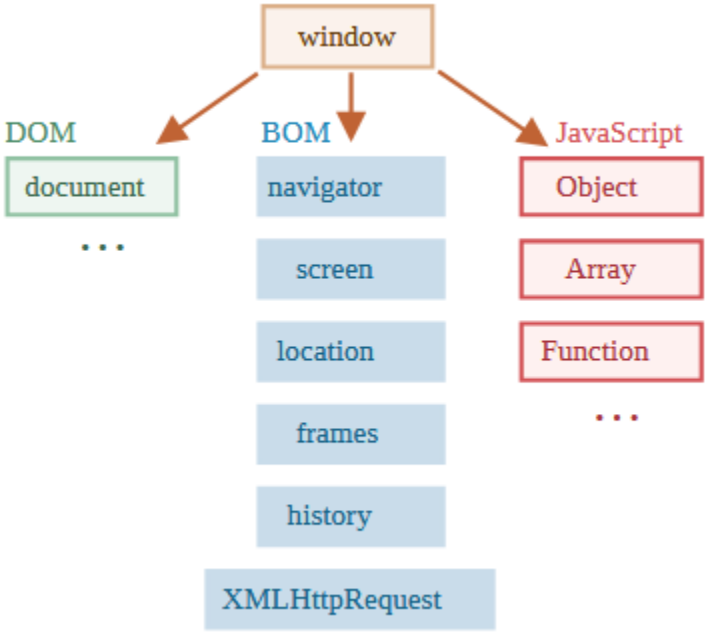
Работа с AJAX и DOM с использованием языка JavaScript: методические указания к выполнению практикума № 4 и лабораторной работы № 4 по дисциплине «Языки интернет- программирования» / В. И. Кузнецов.

Описана теория по DOM, AJAX и BOM. Приведены примеры кода и хорошие практики разработки.

Для студентов МГТУ им. Н.Э. Баумана, обучающихся по направлению «Информатика и вычислительная техника».

Теоретическая часть

Первичной целью языка программирования JavaScript было «оживление» веб-страниц из чего у данный язык обладает огромным функционалом взаимодействия с браузерами и языки разметки HTML и CSS. JavaScript выполняемый в браузере обладает следующим функционалом.



Window является корневым глобальным объектом дающий доступ к API браузера разделенным на 3 категории:

- DOM – взаимодействие с html и css
- BOM – изолированное взаимодействие только с браузером
- CSSOM – отдельная модель взаимодействия с CSS из-за особенностей объектной модели.

Document Object Model

Document Object Model, сокращённо DOM – объектная модель документа, которая представляет все содержимое страницы в виде объектов, которые можно менять. Объект document – основная

«входная точка». С его помощью мы можем что-то создавать или менять на странице.

Основой HTML-документа являются теги. В соответствии с объектной моделью документа («Document Object Model», коротко DOM), каждый HTML-тег является объектом. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом. Все эти объекты доступны при помощи JavaScript, мы можем использовать их для изменения страницы.

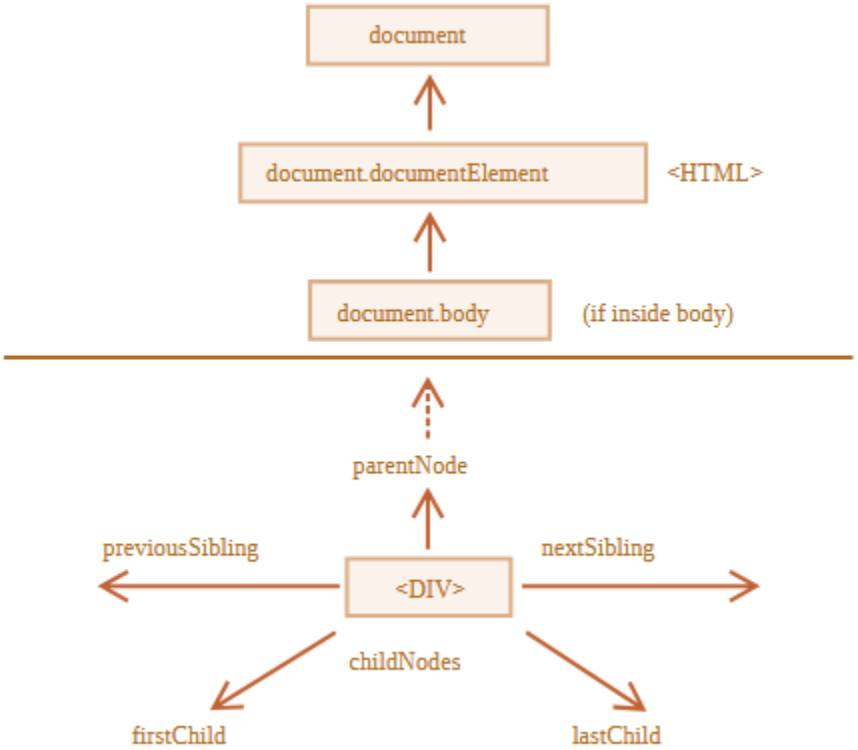
Например, document.body – объект для тега <body> .

DOM – это представление HTML-документа в виде дерева тегов. Каждый узел в этом дереве представляет собой объект. Теги выступают в роли узлов-элементов (или просто элементов) и формируют

древовидную структуру: тег <html> является корневым узлом, а <head> и <body> — его дочерними элементами и так далее. Текст, находящийся внутри элементов, образует текстовые узлы, которые обозначаются как #text. Такой узел содержит только строку текста и не может иметь дочерних элементов — он всегда располагается на самом нижнем уровне дерева.

Все, что есть в HTML, даже комментарии, является частью DOM. Даже директива <!DOCTYPE...> , которую мы ставим в начале HTML, тоже является DOM-узлом. Она находится в дереве DOM прямо перед <html> . Мы не будем рассматривать этот узел, мы даже не рисуем его на наших диаграммах, но он существует.

DOM позволяет нам делать что угодно с элементами и их содержимым, но для начала нужно получить соответствующий DOM-объект. Все операции с DOM начинаются с объекта document . Это главная «точка входа» в DOM. Из него мы можем получить доступ к любому узлу.



Поиск, чтение и изменение

Чаще всего используют querySelector/All и getElementById. Для контента: textContent — безопасно для текста; innerHTML — с осторожностью (риск XSS). Для атрибутов: getAttribute/setAttribute и dataset для data-* атрибутов.

```
const el = document.querySelector('#title');
el.textContent = 'Новый заголовок';     // безопасно
el.setAttribute('aria-live', 'polite');
console.log(el.dataset.role);           // доступ к data-role
```

Классы, стили и метрики

```
const box = document.querySelector('.box');
box.classList.toggle('active');
box.style.transition = 'opacity .2s';
const { width, height, top } = box.getBoundingClientRect();
console.log(width, height, top);
```

Создание и массовые вставки

Для производительности используйте DocumentFragment, шаблонные элементы <template>, батчинг обновлений.

```
const list = document.querySelector('#list');
const frag = document.createDocumentFragment();
for (let i = 0; i < 200; i++) {
  const li = document.createElement('li');
  li.textContent = 'Элемент ' + (i + 1);
  frag.append(li);
}
list.append(frag);
```

Наблюдение за изменениями (MutationObserver)

MutationObserver позволяет реагировать на изменения дерева без периодических опросов.

```
const target = document.querySelector('#dynamic');
const mo = new MutationObserver((list) => {
  for (const m of list) {
    console.log('Изменение:', m.type, m);
  }
});
mo.observe(target, { childList: true, subtree: true, attributes: true
});
```

Browser object model

Браузер предоставляет свой объект, а также отдельные стандартные функции. С помощью них можно организовывать переходы между страницами, вывод модальных окон и тд.

Browser API

alert

С этой функцией мы уже знакомы. Она показывает сообщение и ждёт, пока пользователь нажмёт кнопку «ОК». Это небольшое окно с сообщением называется *модальным окном*. Понятие *модальное* означает, что пользователь не может взаимодействовать с интерфейсом остальной части страницы, нажимать на другие кнопки и т.д. до тех пор, пока взаимодействует с окном. В данном случае – пока не будет нажата кнопка «ОК».

prompt

Функция prompt принимает два аргумента:

```
result = prompt(title, [default]);
```

Этот код отобразит модальное окно с текстом, полем для ввода текста и кнопками OK/Отмена. Title - Текст для отображения в окне. Default - Необязательный второй параметр, который устанавливает начальное значение в поле для текста в окне.

```
result = prompt(title, [default]);
```

Пользователь может напечатать что-либо в поле ввода и нажать OK. Введённый текст будет присвоен переменной result. Пользователь также может отменить ввод нажатием на кнопку «Отмена» или нажав на клавишу Esc. В этом случае значением result станет null.

confirm

```
result = confirm(question);
```

Функция confirm отображает модальное окно с текстом вопроса question и двумя кнопками: OK и Отмена.

Результат – true, если нажата кнопка OK. В других случаях – false.

BOM (Browser Object Model)

Объектная модель браузера (Browser Object Model, BOM) – это дополнительные объекты, предоставляемые браузером (окружением), чтобы работать со всем, кроме документа.

Например:

- Объект navigator даёт информацию о самом браузере и операционной системе. Среди множества его свойств самыми известными являются: navigator.userAgent – информация о текущем браузере, и navigator.platform – информация о платформе (может помочь в понимании того, в какой ОС открыт браузер – Windows/Linux/Mac и так далее).
- Объект location позволяет получить текущий URL и перенаправить браузер по новому адресу.

```
alert(location.href); // показывает текущий URL

if (confirm("Перейти на Wikipedia?")) {

  location.href = "https://wikipedia.org"; // перенаправляет
браузер на другой URL

}
```

AJAX: подробности Fetch API, кэширование, потоки, безопасность

Fetch API: базовые паттерны

```
async function getJSON(url) {
  const res = await fetch(url);
  if (!res.ok) throw new Error('HTTP ' + res.status);
  // Ошибки парсинга JSON тоже ловим отдельно
  try { return await res.json(); }
  catch (e) { throw new Error('Некорректный JSON'); }
}
```

Методы и типы тела запроса

```
// Отправка JSON
await fetch('/api/items', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ title: 'Test' })
});

// Отправка формы / файлов
const form = document.querySelector('form');
const data = new FormData(form);
await fetch('/upload', { method: 'POST', body: data });
```

Параметры fetch: mode, credentials, cache, redirect, referrerPolicy

```
await fetch('/endpoint', {
  mode: 'cors',                // 'cors' | 'no-cors' | 'same-origin'
  credentials: 'include',      // 'omit' | 'same-origin' | 'include'
  cache: 'no-store',           // 'default' | 'no-store' | 'reload' |
'no-cache' | 'force-cache' | 'only-if-cached'
  redirect: 'follow',          // 'follow' | 'error' | 'manual'
  referrerPolicy: 'strict-origin-when-cross-origin',
});
```

mode управляет междоменным поведением; credentials — передачей cookies/авторизации; cache — взаимодействием с HTTP-

кешем; redirect определяет реакцию на редиректы; referrerPolicy — какой реферер отправлять.

Обработка ошибок и таймауты

```
function withTimeout(promise, ms=5000) {
  const ctrl = new AbortController();
  const t = setTimeout(() => ctrl.abort(), ms);
  return {
    signal: ctrl.signal,
    wrap: promise.finally(() => clearTimeout(t))
  };
}

const url = '/api/data';
const { signal, wrap } = withTimeout(fetch(url, { signal: undefined }),
3000);
// Пример с объединением:
fetch(url, { signal }).then(r => r.json()).then(console.log)
.catch(e => {
  if (e.name === 'AbortError') console.warn('Таймаут');
  else console.error('Ошибка сети/HTTP', e);
});
```

Стриминг ответов (ReadableStream)

Потоковая обработка полезна для больших ответов и форматов NDJSON.

```
async function streamLines(url) {
  const res = await fetch(url);
  if (!res.body) throw new Error('Стрим недоступен');
  const reader = res.body.getReader();
  const decoder = new TextDecoder();
  let buf = '';
  while (true) {
    const { value, done } = await reader.read();
    if (done) break;
    buf += decoder.decode(value, { stream: true });
    let idx;
    while ((idx = buf.indexOf('\n')) >= 0) {
      const line = buf.slice(0, idx); buf = buf.slice(idx + 1);
      if (line.trim()) console.log('LINE:', line);
```

```
    }
  }
}
```

Кэширование: заголовки и Cache API (вкратце)

HTTP-заголовки Cache-Control/ETag/Last-Modified управляют повторным использованием ответов. Cache API позволяет программно кешировать запросы в сервис-воркерах. Для простых случаев достаточно параметра fetch.cache.

```
// Пример ручного обхода повторных запросов (эвристика)
const memo = new Map();
async function cachedJSON(url) {
  if (memo.has(url)) return memo.get(url);
  const p = fetch(url).then(r => r.json());
  memo.set(url, p);
  return p;
}
```

Безопасность: CORS, XSS, CSRF (основы)

- CORS определяет, какие источники могут читать ответы. Если браузер блокирует — проверьте заголовки ответа (Access-Control-Allow-Origin и др.).
- XSS: никогда не вставляйте ответ сервера в innerHTML без очистки. Для текста — используйте textContent.
- CSRF: запросы, использующие cookies, должны защищаться со стороны сервера (токены/заголовки). На клиенте — минимизируйте «скрытые» автоматические отправки.

XMLHttpRequest: прогресс загрузки

```
function upload(file) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
```

```
    xhr.open('POST', '/upload');
    xhr.upload.onprogress = (e) => {
      if (e.lengthComputable) {
        console.log('Прогресс:', Math.round(e.loaded / e.total * 100) +
'%');
      }
    };
    xhr.onload = () => xhr.status >= 200 && xhr.status < 300
      ? resolve(xhr.responseText)
      : reject(new Error('HTTP ' + xhr.status));
    xhr.onerror = reject;

    const form = new FormData();
    form.append('file', file);
    xhr.send(form);
  });
}
```

Альтернативы AJAX (для справки)

- Server-Sent Events — однонаправленные события от сервера к клиенту.
- WebSocket — двунаправочный постоянный канал связи.

Хранилища и события storage

```
localStorage.setItem('k', 'v');
console.log(localStorage.getItem('k'));
window.addEventListener('storage', (e) => {
  console.log('Изменение в другой вкладке:', e.key, e.newValue);
});
```

Cookies (клиентская сторона)

Cookies читаются/записываются через document.cookie. Флаги Secure/SameSite/Lax/Strict устанавливаются сервером. Cookies с флагом HttpOnly недоступны из JavaScript (это нормально и безопасно).

```
// Простая запись (дата истечения в UTC-формате)
document.cookie = "theme=dark; path=/; max-age=31536000";
```

Межвкладочное взаимодействие и безопасность

postMessage позволяет безопасно отправлять сообщения между окнами/вкладками разных источников, если указать целевой origin.

```
// В окне-отправителе:
otherWindow.postMessage({ type: 'ping' }, 'https://example.com');
// В окне-получателе:
window.addEventListener('message', (e) => {
  if (e.origin !== 'https://example.com') return;
  console.log('Сообщение:', e.data);
});
```

Жизненный цикл страницы и производительность

- document.visibilityState и событие visibilitychange для паузы фоновых задач.
- requestIdleCallback — выполнение низкоприоритетных задач, когда браузер свободен.
- Performance API: performance.now(), performance.mark/measure для измерений.

Практическая часть

Как запускать код в браузере

1)Создайте файл index.html и подключите к нему скрипт script.js через тег <script>.

2) Откройте index.html двойным кликом — страница откроется в браузере.

3) Откройте DevTools (F12) → Console/Network/Sources для отладки.

Минимальный шаблон HTML

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>DOM & AJAX</title>
</head>
<body>
  <h1 id="title">Привет, DOM!</h1>
  <button id="btn">Нажми меня</button>

  <script src="script.js" defer></script>
</body>
</html>
```

Файл script.js

```
document.addEventListener('DOMContentLoaded', () => {
  const btn = document.getElementById('btn');
  btn.addEventListener('click', () => alert('Кнопка нажата!'));
});
```

Типичные ошибки и отладка

- Неправильные селекторы или обращение к элементам до DOMContentLoaded.
- Опасная вставка через innerHTML для непроверенных данных (XSS). Предпочитайте textContent.
- Забытый e.preventDefault() в обработчике submit приводит к перезагрузке.
- Fetch: отсутствие проверки res.ok и обработки JSON-парсинга.

- Таймауты не реализованы: используйте AbortController.
- Смешанный контент (HTTP в HTTPS-странице) блокируется браузером.
- Кросс-доменные ошибки (CORS) неверно диагностируются — проверяйте вкладку Network и заголовки.
- Лишние reflow: чередование чтений/записей стилей, отсутствие батчинга.
- Игнорирование accessibility: отсутствуют aria-* и фокус-менеджмент.

Инструменты отладки

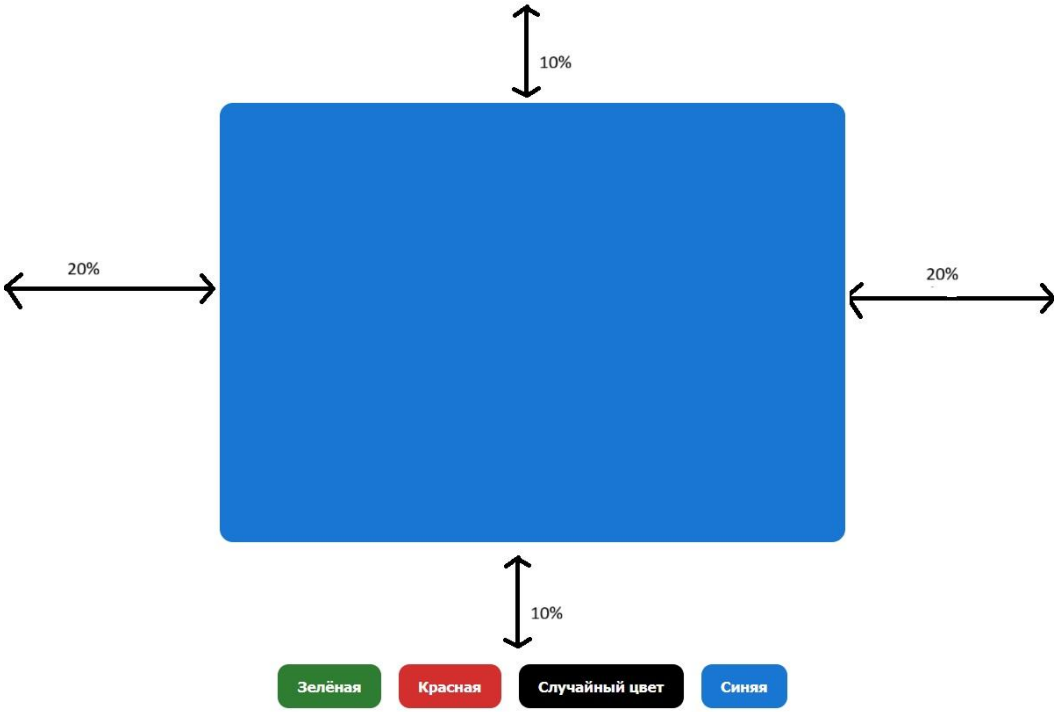
- Console: console.log/table/time/error; Source maps и debugger.
- Network: статусы, заголовки, предпросмотр ответов, CORS, тайминги.
- Performance: записи профиля, Layout/Rendering, FPS.
- Lighthouse: базовая проверка производительности и доступности.

Задания

Задание 1. Работа с CSS через JavaScript

Сверстать приведённую ниже страницу и реализовать логику для кнопок. При нажатии на:

- зеленая кнопка меняет цвет блока на зеленый
- синяя кнопка меняет цвет блока на синий
- случайный цвет кнопка меняет цвет блока на любой случайный цвет
- красная кнопка меняет цвет блока на красный



Задание 2. Расписание

Взять из 3 пункта 2 лабораторной работы код расписания и сделать расписание интерактивным. При наличии информации в поле то новая должна пере затирать старую. Отчистка поля должна быть сделана через обновление предмета без указания названия и информация в поле должна стираться полностью.

Контрольные вопросы

1. Чем отличаются Node, Element и Text в DOM и почему это важно при обходе дерева?
2. Объясните разницу между DOMContentLoaded и load. Когда какое событие использовать?
3. Как делегирование событий уменьшает количество обработчиков и влияет на производительность?
4. Какие параметры fetch (mode, credentials, cache, referrerPolicy) вы настраивали бы для междоменного запроса и почему?
5. В чём преимущества потоковой обработки ответов (ReadableStream) и какие форматы для этого подходят?
6. Какие интерфейсы BOM отвечают за навигацию и состояние истории (location, history) и как ими пользоваться без перезагрузки страницы?
7. Как безопасно работать с данными из сети при рендере в DOM, чтобы избежать XSS, и какую роль играет CORS?