Министерство образования Республики Беларусь
Белорусский государственный университет
Механико-математический факультет
Кафедра веб-технологий и компьютерного моделирования

СОГЛАСОВАНО
Заведующий кафедрой
_____ М.В. Игнатенко
«22» сентября 2025 г.

СОГЛАСОВАНО
Декан факультета
_____ С.М. Босяков
«30» сентября 2025 г.

Веб-программирование

Электронный учебно-методический комплекс для специальности:
6-05-0533-07 «Математика и компьютерные науки»,
профилизация «Математика»,
6-05-0533-08 «Компьютерная математика и системный анализ»

Регистрационный № 2.4.3-24 / 672

Автор:
Барвенов С.А., кандидат физ.-мат. наук, доцент

Рассмотрено и утверждено на заседании Совета Механико-математического факультета. Протокол № 1 от 30.09.2025 г.

Минск 2025

УДК 004.42:004.738.5(075.8)
Б 246

Утверждено на заседании Научно-методического совета БГУ
Протокол № 3 от 30.10.2025 г.

А в т о р :

Барвенов Сергей Александрович, кандидат физико-математических наук, доцент, доцент кафедры кафедра веб-технологий и компьютерного моделирования механико-математического факультета БГУ.

Рецензенты:

кафедра информационных технологий факультета цифровой экономики Белорусского государственного экономического университета (заведующий кафедрой Садовская М.Н., кандидат технических наук, доцент);

Матейко О.М., кандидат физико-математических наук, доцент кафедры общей математики и информатики Механико-математического факультета Белорусского государственного университета.

Барвенов, С. А. Веб-программирование : электронный учебно-методический комплекс для специальностей 6-05-0533-07 «Математика и компьютерные науки», 6-05-0533-08 «Компьютерная математика и системный анализ» / С. А. Барвенов ; БГУ, Механико-математический фак., Каф. веб-технологий и компьютерного моделирования. – Минск : БГУ, 2024. – 267 с. : ил., табл. – Библиогр.: с. 264–265.

Электронный учебно-методический комплекс (ЭУМК) по учебной дисциплине «Веб-программирование» предназначен для студентов специальности 6-05-0533-07 « Математика и компьютерные науки». Профилизация: «Математика» и 6-05-0533-08 «Компьютерная математика и системный анализ».

ЭУМК содержит тексты лекций, материалы для лабораторных занятий, перечень контрольных вопросов, списки рекомендованной литературы.

СОДЕРЖАНИЕ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Электронный учебно-методический комплекс (ЭУМК) по учебной дисциплине «Веб-программирование» предназначена для студентов специальности 6-05-0533-07 «Математика и компьютерные науки» Профилизация: «Математика» и специальности 6-05-0533-08 «Компьютерная математика и системный анализ».

Содержание разделов ЭУМК соответствует образовательным стандартам, структуре и тематике примерного учебного плана № 6-05-05-028/пр. от 31.01.2023 г, учебных планов БГУ: №6-5.4-55/01 от 15.05.2023, №6-5.4-55/11ин. от 31.05.2023, №6-5.4-56/01 от 15.05.2023, №6-5.4-56/11ин. от 31.05.2023г.

Комплекс подготовлен в соответствии с требованиями Положения обучебно-методическом комплексе на уровне высшего образования, утвержденного Постановлением министерства образования Республики Беларусь от 26.07.2011 № 167.

Учебно-методический комплекс преследует **цель** по оказании посильной помощи студентам в овладевании современными методами и технологиями проектирования сайтов и их оформления, подготовке квалифицированных педагогических кадров, способных на практике применить современные информационные веб-технологии в своей будущей педагогической деятельности.

В структуру ЭУМК входит:

1. Теоретический раздел (включает конспект лекций по учебной дисциплине).

2. Практический раздел.

3. Раздел контроля знаний (содержит перечень контрольных мероприятий управляемой самостоятельной работы студентов, вопросы для подготовки к зачету).

4. Вспомогательный раздел (включает список литературы).

Работа студента с ЭУМК должна включать ознакомление с тематическим планом учебной дисциплины, представленным в учебной программе учреждения высшего образования, в которой можно получить информацию о тематике лекций, лабораторных занятий и рекомендуемой литературе. Для подготовки к лабораторным занятиям рекомендуется использовать материалы, представленные в теоретическом и практическом разделах ЭУМК, а также материалы для контроля самостоятельной работы студентов. В ходе подготовки к зачету целесообразно ознакомиться с требованиями к компетенциям по учебной дисциплине, изложенными в учебной программе учреждения высшего образования, а также перечнем вопросов к зачету и экзамену.

Задачи учебной дисциплины

- получение представления о тенденциях и направлениях в развитии современных информационных технологий;
- дать практические навыки использования языка HTML5 и спецификации CSS3;
- изучения некоторых библиотек верстки;

- изучение основ JavaScript;
- изучение библиотеки React;
- изучение основ PHP.
- получение студентами практических навыков по конструированию сайтов.

Место учебной дисциплины в системе подготовки специалиста с высшим образованием.

Учебная дисциплина «Веб-программирование» относится к **модулю «Программирование»** компонента учреждения образования специальности 6-05-0533-08 Компьютерная математика и системный анализ, государственного компонента специальности 6-05-0533-07 Математика и компьютерные науки..

Учебная дисциплина «Веб-программирование» взаимосвязана с учебной дисциплиной «Программирование» государственного компонента: «Методы программирования» и «Технологии программирования».

Требования к компетенциям

Освоение учебной дисциплины должно обеспечить формирование у студентов следующих компетенций:

универсальные компетенции:

компетенции, предусмотренные для специальности 6-05-0533-07

универсальные компетенции:

Решать стандартные задачи профессиональной деятельности на основе применения информационно-коммуникационных технологий.

профессиональные компетенции:

Применять современные технологии и базовые конструкции языков программирования для реализации алгоритмических прикладных задач и разработки веб-проектов.

компетенции, предусмотренные для специальности 6-05-0533-08

универсальные компетенции:

Решать стандартные задачи профессиональной деятельности на основе применения информационно-коммуникационных технологий.

специализированные компетенции:

Применять современные технологии и базовые конструкции языков программирования, проектировать, создавать и использовать базы данных для реализации алгоритмических прикладных задач и разработки веб-проектов.

В результате освоения учебной дисциплины студент должен:

знать:

- основные понятия, терминологию и структуру компьютерных сетей и интернет;
- язык HTML5, основные возможности CSS3;
- основы JavaScript;
- основы программирования серверной части веб-приложений;

уметь:

- проектировать интерфейс веб-приложений;
- верстать веб-страницы, создавать и использовать CSS;

- создавать динамические страницы средствами JavaScript;
- использовать технологии взаимодействия с сервером на своей странице;
- программировать обработку данных, переданных с html страницы на сервере;

иметь навык:

- приемов проектирования веб-приложений;
- работы с инструментами разработки дизайна сайтов;
- работы в “инструментах разработчика” основных браузеров;
- работы со средствами разработки клиентских и серверных частей сайта.

Структура учебной дисциплины

Дисциплина изучается во 5 семестре. В соответствии с учебным планом всего на изучение учебной дисциплины «Веб-программирование» **для очной формы** получения высшего образования – 108 часов *для специальности 6-05-0533-07*, в том числе 72 аудиторных часов, 110 часов *для специальности 6-05-0533-08,* в том числе 72 аудиторных часов. Из них:

Специальность 6-05-0533-07 Математика и компьютерные науки:

Лекции – 36 часа, лабораторные занятия – 30 часов, управляемая самостоятельная работа (УСР) – 6 часов.

Трудоемкость учебной дисциплины составляет 3 зачетные единицы.

Форма промежуточной аттестации – зачет.

Специальность 6-05-0533-08 Компьютерная математика и системный анализ:

Лекции – 36 часа, лабораторные занятия – 32 часов, управляемая самостоятельная работа (УСР) – 4 часа.

Трудоемкость учебной дисциплины составляет 3 зачетные единицы.

Форма промежуточной аттестации – зачет.

1. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

В теоретическом разделе изложен краткий конспект лекций в соответствии с программой дисциплины.

1.1. Интернет. Основные понятия

В настоящее время не вызывает никаких сомнений необходимость в изучении технологий, связанных с созданием не просто статических сайтов , а веб-приложений. Трудно найти область человеческой деятельности, в которой использование таких приложений не принесло бы существенной пользы, поэтому резко повысился спрос на специалистов, способных эффективно разрабаитывать их. Соответственно появилась необходимость в обучении студентов современным информационным веб-технологиям, и подготовке их к профессиональной практической работе.

1.1.1. Основные понятия представления, организации и передачи информации в Интернете

Устройства, подключенные к сети, часто условно разделяют на клиенты и серверы.

Клиенты – устройства, например, планшеты, компьютеры или телефоны, которые выходят в Интернет с использованием программного обеспечения доступного на этих устройствах (как правило, браузеров или мобильных приложений).

Серверы – это компьютеры, с специальным програмным обсепечением, которые хранят статические веб-страницы, сайты или работающие приложения, которые генерируют страницы и «отдают» информацию.

Упрощенная схема того, как они взаимодействуют по протоколу HTTP, может выглядеть как на рисунке 1. Отметим, что есть и другие протоколы и подходы для реализации связи между сервером и кленнтом в реальном времени: WebSocket, Server-Sent Events (SSE), Long Polling.

Когда клиентское устройство пытается получить доступ к ресурсу, данные загружаются с сервера на клиентский компьютер для отображения в браузере пользователя.

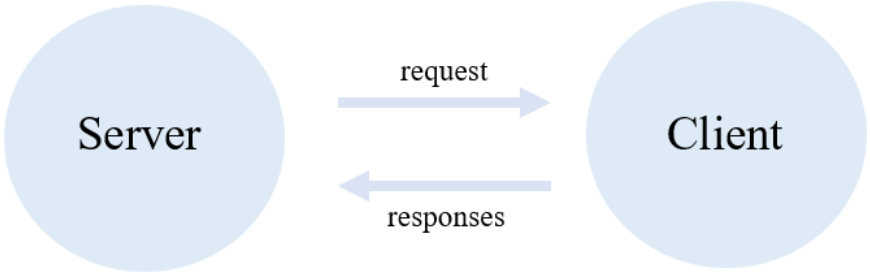


Рисунок 1 – Упрощенная схема взаимодействия.

TCP/IP (Transmission Control Protocol / Internet Protocol) сетевая модель передачи данных, представленных в цифровом виде. Протокол являются коммуникационным и определяет, каким образом данные должны передаваться по сети.

DNS (Domain name server) – система доменных имен. У любого устройства в сети есть свой числовой адрес, он называется **IP-адрес** (Internet Protocol address). Реальные веб-адреса представляют собой строки чисел, например, 173.194.121.32 (псевдоним этого IP-адреса доменное имя google.com). Такой набор чисел представляет собой уникальное местоположение в Интернете. Запоминать такие адреса тяжело, поэтому используют DNS. DNS-сервера связывают символьный веб-адрес, который вы вводите в браузере, например, bsu.by, с реальным IP-адресом сайта.

URL (Uniform Resource Locator) – адрес ресурса в сети, определяет местонахождение и способ обращения к нему. Например:

- https://bsu.by/universitet.php;
- ftp://examples.com/workbook/download.zip;
- mailto:Ivanov@mail.ru;
- file:///home/user/letter.txt;

URI (Uniform Resource Identifier) – имя и адрес ресурса в сети [**URN** + **URL**]**.** Например, адрес сайта это URI: http://google.com.

Если рассмотреть адрес сайта https://www.example.com/folder/page.html, то

- URI – https://www.example.com/folder/page.html;
- URL - https://www.example.com;

HTTP (HyperText Transfer Protocol или Протокол Передачи Гипертекста) – это протокол, который определяет язык для клиентов и серверов, чтобы общаться друг с другом.

Пакеты. В основном, когда данные передаются через Интернет, они отправляются в виде тысяч мелких кусочков, так что множество разных пользователей могут скачивать один и тот же сайт одновременно. Если бы данные отправлялись одним большим куском, тогда только один пользователь мог бы скачать его за один раз, и это сделало бы пользование интернетом не эффективным.

Маршрутизатор – специализированное устройство, которое пересылает пакеты между различными сегментами сети на основе правил и таблиц маршрутизации.

Файлы компонентов сайта. Обычно сайт состоит из нескольких различных файлов, которые бывают двух основных типов: файлы кода (HTML, CSS и JavaScript и др.) и материалы (изображения, музыка, видео, документы Word и PDF и т.д.).

Когда вы вводите веб-адрес в свой браузер происходит следующая последовательность действий:

1. Браузер обращается к DNS серверу и находит реальный адрес сервера, на котором расположен сайт.

2. Браузер посылает HTTP запрос к серверу, запрашивая его отправить копию сайта для клиента. Это сообщение и все остальные данные, передаваемые между клиентом и сервером, передаются по интернет-соединению с использованием протокола TCP/IP.
3. Если сервер одобряет запрос клиента, сервер отправляет клиенту статус «200 OK», и затем начинает отправку файлов сайта в браузер в виде небольших порций, называемых пакетными данными.
4. Браузер собирает маленькие куски (пакеты) в полноценный сайт и показывает его в браузере.

1.1.2. Фронтенд и бэкенд

Раньше сайты представляли собой набор статических страниц с табличной разметкой. Но Веб-разработка не стоит на месте. Появились новые языки и технологии разработки сайтов.

Самая простейшая веб-страница в браузере представляется для пользователя как текст, стилизованный определенным образом. Сложная и многоуровневая структура современных веб-приложений требует иерархического разделения процесса их разработки. Исторически этот процесс разделяется на две части: ***front-end*** (клиентскую) и ***back-end*** (серверную). Рассмотрим их более подробно. Эти термины имеет три различных значения.

Первое значение: **фронтенд – браузер**, **бэкенд – сервер**. В веб-разработке в качестве фронтенда выступают html-верстка, стили CSS и JavaScript, а в качестве бэкенда – серверная часть, которую программируют, например, на JavaScript под Node.js, PHP, java или ASP.net. Грубо говоря, все то, что исполняется на стороне клиента – front-end, а то, что на стороне сервера – back-end.

Второе значение: **фронтенд – статика**, **бэкенд – программный код**. В среде разработчиков высоконагруженных систем (highload-разработчиков) термином front-end называют ту программную часть, которая непосредственно «отдаёт» контент. Например, на больших проектах часто программную серверную часть представляют два веб-сервера – Apache и Nginx. Nginx принимает запросы и, в случае статического файла, (изображение, файл css, js или xml) сразу же отдаёт его содержимое, а в случае, например, PHP-скрипта, отправляет его к серверу Apache, который уже умеет обрабатывать PHP. Тут Nginx – это фронтенд, а Apache – бэкенд.

Третье значение: **фронтенд – открытые данные**, **бэкенд – административная часть**. Если речь идет о CMS (Content Management System или Система управления контентом) административную часть называют back-end, а клиентскую часть сайта – front-end.

В рамках данного курса будем использовать первое значение: фронтенд – это все то, что исполняется на стороне клиента. А значит, к фронтенд-технологиям можно отнести весь пласт языков и технологий для создания веб-страниц. Базовыми из них являются: HTML, CSS, JavaScript.

1.1.3. HTML

HTML (HyperText Markup Language) – язык гипертекстовой разметки документов. Размеченный документ определенным образом интерпретируется браузерами и отображается в человекочитаемой форме.

HTML позволяет:

создавать обычные текстовые документы;

структурировать и форматировать текстовые документы;

соединять текстовые документы между собой через гиперссылки;

вставлять в текстовые документы рисунки, графику, аудио и видео контент.

Разберем термин HTML.

Гипертекст (HyperText) означает интерактивный текст, т.е. текст, с которым пользователь может взаимодействовать. Термин «гипертекст» ввёл Теодор Нельсон в работе «A File Structure for the Complex, the Changing and the Indeterminate», опубликованной в 1965 году. Традиционный принцип расположения текста линейный: слово следует за словом, фраза за фразой, страница следует за страницей, читать нужно слева направо. Нельсон же предложил располагать информацию в узлах сети, связь между которыми не жесткая, а выбирается самим пользователем.

Разметка (Markup**)** обозначает возможность создавать контент в различных формах: заголовок, параграф, список, таблица, рисунок, ссылка и т.д., чтобы затем определять различные атрибуты для этих элементов контента.

Язык (Language) – это язык, который имеет свой собственный синтаксис, словарь и правила написания.

1.1.4. CSS

CSS (Cascading Style Sheets) – язык, описывающий внешний вид (оформление) и поведение html-элементов. Он позволяет применять стили выборочно к элементам в документах HTML. Дословно CSS переводится как каскадные таблицы стилей. Пример влияния CSS на вид страницы хорошо иллюстрирует сайт «Сад CSS-дзена» http://www.csszengarden.com/tr/ru/, который демонстрирует одну и ту же html-страницу, к которой по ссылкам внутри подключаются различные стили оформления. При этом представление контента меняется неузнаваемо. На сайте можно скачать эти css-файлы для последующего их использования. Ниже на рисунках 2-4 приведены несколько примеров оформления с сайта «Сад CSS-дзена».

На сайте w3schools.com также есть прекрасная демонстрация применения различных стилей к простейшей странице https://www.w3schools.com/css/css_intro.asp.

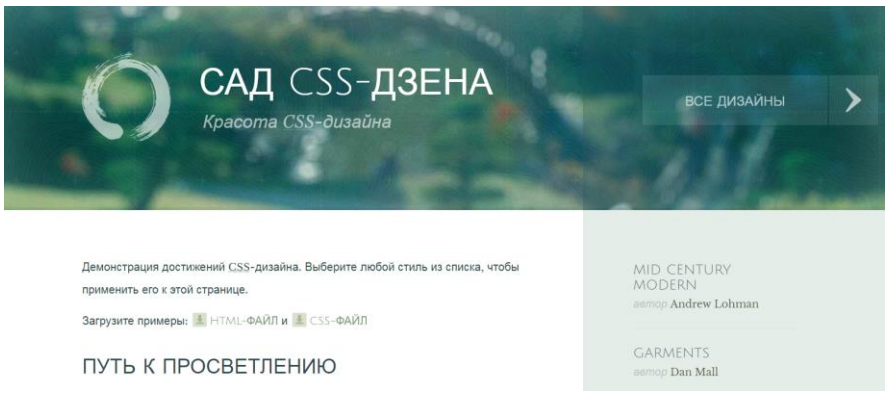


Рисунок 2 – Вариант дизайна «классический».

Рисунок 3 – Вариант дизайна «пресса».

Рисунок 4 – Вариант дизайна «старая аптека».

Очень популярен фреймворк Bootstrap (http://getbootstrap.com/, http://mybootstrap.ru/) – свободный набор инструментов для стилизации сайтов и веб-приложений. Bootstrap включает в себя html- и css-шаблоны оформления для типографики, веб-форм, кнопок, меток, блоков навигации и прочих компонентов веб-интерфейса, включая JavaScript-расширения.

1.1.5. JavaScript

JavaScript (JS) – это язык программирования, который применяется в html-документах, и может обеспечить динамическую интерактивность на странице. Встроен во все браузеры. Язык применяется в совокупности с различными библиотеками (jQuery, LoDash и др.) и фреймворками (Vue.js, React.js, AngularJS, Ember, NodeJS и др.), которые облегчают решение определенных, часто встречающихся задач. Кроме этого уже разработано большое количество

инструментов с использованием основного языка JavaScript. К ним относятся в частности программные интерфейсы приложения или API. Есть API встроенные в браузеры, обеспечивающие различные функциональные возможности, такие как динамическое создание HTML и установка CSS стилей, захват и манипуляция видеопотоком, работа с веб-камерой пользователя или генерация 3D графики. Есть сторонние API, которые позволяют разработчикам внедрять в свои сайты функциональность, разработанную другими программистами.

JavaScript Object Notation (JSON) – это формат обмена данными. По своему синтаксису этот формат напоминает JavaScript. JSON поддерживают множество языков программирования, но особенно он полезен для JavaScript-приложений, таких как веб-сайты и расширения для браузера.

1.1.6. Библиотека jQuery

jQuery – очень популярная библиотека JavaScript, которая используется во многих проектах, которые созданы давно, но продолжают развиваться. Она была создана для удобного взаимодействия JavaScript и HTML. Библиотека jQuery помогала (когда подобных методов еще не было в Js) легко получать доступ к любому элементу DOM (Document Object Model), обращаться к их атрибутам и содержимому, манипулировать ими. Также библиотека jQuery предоставляет API для работы с AJAX.

1.1.7. AJAX

AJAX (Asynchronous JavaScript and XML)– это не технология сама по себе, а термин, который описывает подход к совместному использованию существующих технологий в 2000-х годах. AJAX включает: HTML, JavaScript и главное – использование объекта XMLHttpRequest. Когда эти технологии объединяются в модель AJAX, веб-приложения способны делать обновления интерфейса пользователя на странице без необходимости полной перезагрузки страницы браузером. Приложения становятся более отзывчивыми к действиям пользователей.

В данный момент в современной веб-разработке есть более удобные средства для асинхронного взаимодействия с серверами – promise и fetch.

1.1.8. Вспомогательные материалы

Учебные материалы можно найти в сети… Например, консорциума WWW есть собственный обучающий сайт:

http://www.w3schools.com/html/default.asp

На сайте : http://www.htnmlbook.ru тоже есть обучающие материалы по верстке, а на сайте https://learn.javascript.ru/ – уроки и примеры по JS.

Для выполнения обучающих и тестовых заданий, создания примеров удобно использовать песочницы или Online sandboxes. Например, jsbin.com; jsfiddle.net; cssdeck.com; codepen.io.

1.1.9. Необходимые инструменты разработки

Существует определенный минимальный набор средств разработки сайтов и приложений.

Текстовый редактор

Для работы с контентом необходим редактор. Существует два типа редакторов: WYSIWYG и текстовые редакторы HTML. WYSIWYG является аббревиатурой от What You See Is What You Get (Что вы видите, то и получаете). Редакторы этого типа предоставляют интерфейс редактирования, который показывает, как выглядит код на рабочей веб-странице. WYSIWYG-редакторы не требуют знаний HTML, поэтому пользователю, не имеющему опыта программирования, гораздо легче начать работу. Но при этом пользователь не контролирует генерируемый код, и как следствие, код может быть не оптимальным, «тяжелым».

Текстовые редакторы HTML. Чтобы пользоваться этим типом редакторов, нужны знания HTML. Текстовый редактор не всегда даёт возможность предварительно посмотреть, как будет выглядеть живой сайт. Хороший текстовый редактор повышает эффективность работы, а также помогает избежать некоторых наиболее распространенных ошибок.

Текстовый редактор может быть как очень простым, например как Notepad, но лучше использовать например, Notepad++ или VIM, а может обладать более продвинутыми возможностями, например, Visual Studio Code, Sublime Text, Atom, GNU Emacs, WebStorm.

Какой html-редактор подходит именно вам зависит от требований, к продукту, который вы намерены создать с помощью HTML, от вашего текущего уровня знаний. Но в целом, любой разработчик может пользоваться тем, что ему нравится, это не принципиально.

Веб-браузеры

Для тестирования кода необходимо иметь несколько веб-браузеров. В настоящее время наиболее часто используемые браузеры это Firefox, Chrome, Opera, Safari, Microsoft Edge. Необходимо также тестировать сайт на работу на мобильных устройствах и в любых старых браузерах, которые может использовать целевая аудитория сайта, например, IE 6-8. Lynx подходит для того, чтобы увидеть, как сайт воспринимается слабовидящими пользователями.

Графический редактор

В веб-страницы включают изображения. Для их создания и обработки можно использовать графические редакторы, например, Photoshop, The Gimp, Paint.NET или др.

Система контроля версий

Сейчас активно используются репозитории кода с инструментарием контроля версий, позволяющие совместно работать над проектом в команде, обмениваться кодом и избегать редакторских конфликтов, управлять файлами на

сервере. Одним из наиболее популярных является система git и сайты для размещения публичных бесплатных репозиториев: GitHub или GitLab.

Локальный веб-сервер

Некоторые страницы необходимо запускать на веб-сервере. Локальные серверы позволяют запускать свой сайт без использования хостинга, прямо на домашнем компьютере. Локальные сервера используются в процессе разработки и для тестирования. В качестве локального сервера можно использовать OpenServer, WampServer или XAMPP. Также во многие среды разработки (например в VsCode) встроены собственные локальные сервера.

1.1.10. Планирование

Прежде чем создать сайт или веб-страницу надо определиться, какую цель она будет реализовывать, что именно вы хотите, чтобы пользователь сделал на странице: прочитал информацию, нажал кнопку, заполнил форму и т.д. В зависимости от того, что пользователь должен будет сделать на странице, нужно выстраивать дизайн страницы и подбирать контент. Дизайн, имеется в виду структуру и оформление страницы. Нужно взять карандаш и бумагу и сделать примерный набросок страницы. На рисунке 5 приведен пример наброска страницы для выбора путевки в санаторий.

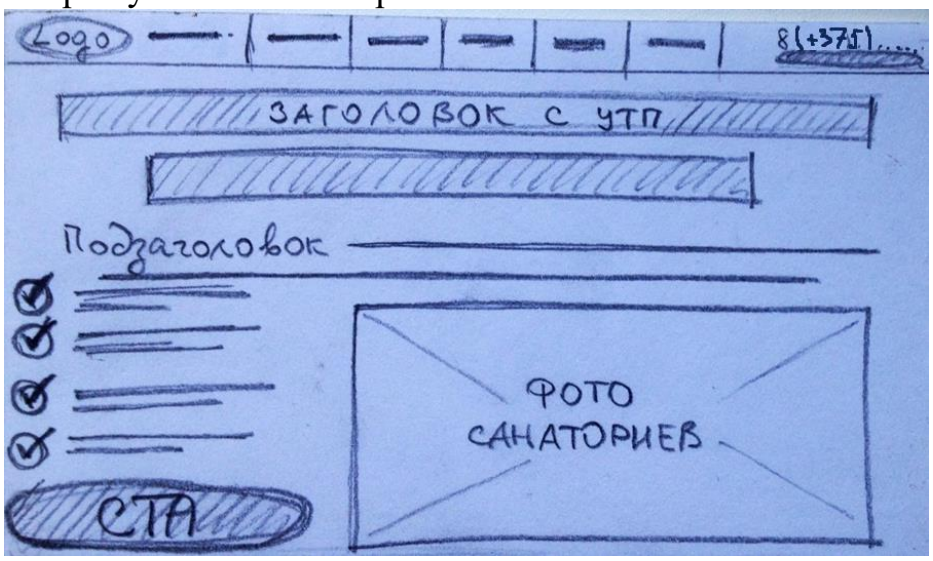


Рисунок 5 – Набросок макета страницы для выбора путевки в санаторий.

Разработка сайта, как правило, начинается именно с набросков страниц на бумаге по которым затем строятся цифровые макеты с использованием графических редакторов или веб-технологий.

Веб-команда крупных реальных проектов включает в себя также графических дизайнеров и дизайнера с опытом взаимодействия, user-experience (UX) designer. Графические дизайнеры работают над визуализацией веб-сайта. UX-дизайнеры разрабатывают концепцию того, как пользователи будут взаимодействовать с веб-сайтом.

1.1.11. Структура сайта на диске

Веб-сайт состоит из множества файлов: текстового контента, кода, стилей, медиа-контента, и т.д. Для удобства структура папок сайта на локальном компьютере должна быть такая же, как и файловая структура опубликованного веб-сайта на сервере.

Все файлы сайта должны находиться к отдельной папке с именем проекта. Наиболее распространенные части, присутствующие в любом проекте сайта – это html-файл главной страницы сайта и папки, содержащие изображения, файлы стилей и файлы скриптов.

Файл с стандратным именем index.html – это главная страница сайта. Название его зависит от требований сервера. Этот файл хранится прямо в корневой папке сайта.

Обычно создают папку images. В ней хранят все изображения, которые используются на сайте. Ее размещают также в корневой папке сайта.

Также в корневой папке сайта создают папки media, styles, scripts. Папка media содержит медиа контент сайта, папка styles – подключаемые таблицы стилей страниц, а папка scripts – весь JavaScript-код, используемый для добавления интерактивных функций на сайте.

Желательно называть папки и файлы полностью в нижнем регистре без пробелов. Регистр важен, поскольку, как правило, веб-серверы, чувствительны к регистру.

Браузеры, веб-серверы и языки программирования не обрабатывают пробелы последовательно. Например, если использовать вставил пробел в имени файла, то некоторые системы могут отнестись к данному имени файла как к двум именам файлов. Некоторые серверы заменяют пробелы в имени файла на "%20" (символьный код для пробелов в URI), в результате чего могут пострадать ссылки. Лучше разделять слова дефисами или нижними подчеркиваниями. Но нижние подчеркивания не всегда хорошо отображаются, поэтому my-file.html смотрится лучше чем my_file.html. Но следует учесть, что поисковая система Google рассматривает дефис как разделитель слов, а знак подчеркивания нет. Поэтому лучше всего писать названия папок и файлов на английском языке в нижнем регистре без пробелов, разделяя слова дефисами. Это позволит в будущем сталкиваться с меньшим количеством проблем из-за совместимости программных продуктов.

1.2. Язык разметки гипертекста HTML

1.2.1. История возникновения HTML

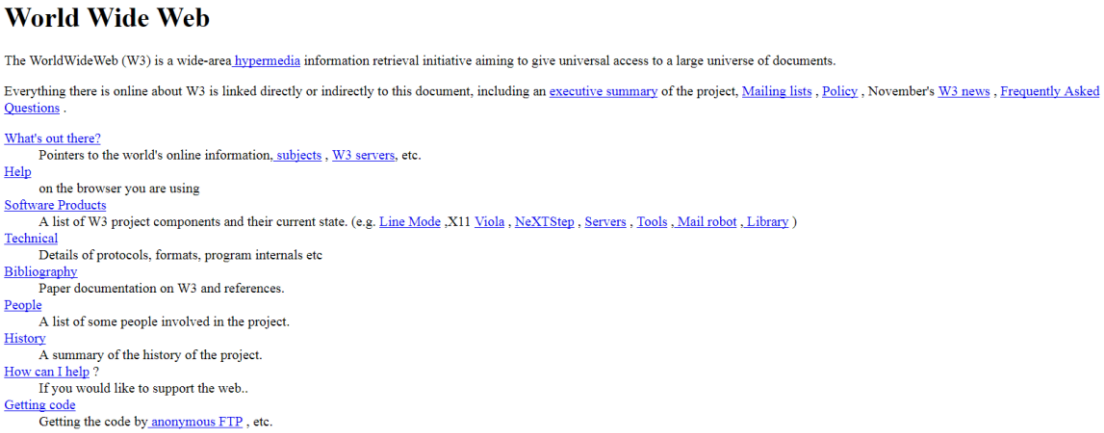
В 60-годы XX века американский ученый Джозеф Карл Робнетт Ликлайдер описал идею сети и назвал ее «Галактическая сеть». В 1969 американское агентство DARPA начало создавать экспериментальную сеть «с коммутацией пакетов». Ее назвали ARPANET. Коммутация пакетов – способ передачи данных по сети, когда информация делится на маленькие пакеты, причем пакеты

отправляются независимо друг от друга. Это обеспечивает надежность, скорость и эффективность пересылки. В декабре 1970 года в Network Working Group придумали протокол управления сетью, а в 1971-1972 его реализовали в ARPANET. Благодаря этому, появилась возможность создавать сетевые приложения. Первым приложением стала электронная почта, ее сделали в 1972 году. Интернет, каким мы его знаем, родился в 1989-1991 годах в Швейцарии в стенах европейского центра ядерных исследований (CERN) в результате работы команды во главе с Тимом Бернерс-Ли и Робертом Кайо над созданием гипертекстовой информационной системы, способной связать воедино бесконечное множество документов. В CERNe были созданы все основные компоненты, из которых состоит веб:

- веб-сервер (назывался HTTPD);
- веб-клиент (первый интернет-браузер с именем World Wide Web – WWW);
- протокол HTTP для обмена между веб-сервером и веб-клиентом;
- язык HTML (стандарт представления документов веб);
- система идентификации документов URI.

Результаты работы были представлены и выложены в глобальной сети для использования на бесплатной основе. Дальнейшей популяризации способствовала разработка браузера Mosaic, разработанного NCSA.

Первый в мире веб-сайт Бернерс-Ли создал по адресу http://info.cern.ch, теперь этот сайт хранится в архиве http://info.cern.ch/hypertext/WWW/TheProject.html. На этом сайте описывалось, что такое Всемирная паутина, как установить веб-сервер, как получить браузер и т. п. Этот сайт также являлся первым в мире интернет-каталогом, потому что позже Тим Бернерс-Ли разместил и поддерживал там список ссылок на другие сайты. На рисунке 6 приведен вид первой веб-страницы из архива.



World Wide Web

The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an executive summary of the project, Mailing lists , Policy , November's W3 news , Frequently Asked Questions .

What's out there?
Pointers to the world's online information, subjects , W3 servers, etc.
Help
on the browser you are using
Software Products
A list of W3 project components and their current state. (e.g. Line Mode ,X11 Viola , NeXTStep , Servers , Tools , Mail robot , Library)
Technical
Details of protocols, formats, program internals etc
Bibliography
Paper documentation on W3 and references.
People
A list of some people involved in the project.
History
A summary of the history of the project.
How can I help ?
If you would like to support the web..
Getting code
Getting the code by anonymous FTP , etc.

Рисунок 6 – Веб-страница из архива

Наибольшее распространение HTML получил в 90-х годах, благодаря ему начался бурный рост популярности Интернета. Первоначально под эгидой различных компаний язык развивался разными путями. Поскольку все разрабатываемые html-документы должны одинаково работать на различных платформах и в различных браузерах, то для унификации практических разработок в 1994 году была создана специальная группа World Wide Web Consortium's HTML

Working Group. Консорциум выпускает спецификации языка. Каждая новая версия HTML пытается добиться еще большей степени универсальности.

Хронология развития HTML представлена в таблице 1:

Таблица 1 – Версии HTML.

Версия	Год
HTML	1991
HTML 2.0	1995
HTML 3.2	1997
HTML 4.01	1999
XHTML	2000
HTML5	2014

Основное отличие HTML5 от предыдущих версий состоит в следующем:

- новые семантические теги;
- был создан новый алгоритм парсинга для создания DOM структуры документа;
- переопределены правила и семантика уже существовавших тегов HTML.

HTML5 можно считать не просто новой версией языка разметки для создания веб-страниц, но фактически он является платформой для создания приложений. HTML5 стал применяться не только для создания веб-страниц, но и для создания мобильных приложений под Android, iOS, Windows Mobile и даже для создания десктопных приложений для обычных компьютеров. Поэтому термин HTML5 используется чаще всего в двух значениях: либо как обновленный язык разметки гипертекста, либо как мощная платформа для создания веб-приложений, которая включает не только непосредственно язык разметки гипертекста, обновленный HTML, но и язык программирования JavaScript и каскадные таблицы стилей CSS3.

1.2.2. W3C. Стандарт HTML

Консорциум Всемирной паутины *(World Wide Web Consortium, W3C)* – независимая международная организация, разрабатывающая и внедряющая технологические стандарты для Всемирной паутины. Консорциум возглавляет сэр Тимоти Джон Бернерс-Ли, автор множества разработок в области информационных технологий. Консорциум определяет стандарт HTML, CSS, HTTP, URI и др. в виде спецификаций. Текущую полную спецификацию на английском языке можно посмотреть по адресу https://www.w3.org/TR/html5/.

HTML разрабатывался с таким расчетом, чтобы все виды устройств, компьютеры с графическими дисплеями различного разрешения и глубины цвета, сотовые телефоны, переносные устройства, разговорные устройства, компьютеры с высокой и низкой тактовой частотой и так далее, могли получать информацию из Интернета. У консорциума есть собственный обучающий сайт: http://www.w3schools.com/html/default.asp.

HTML определяет набор тегов или правил, согласно которым отображается текст в браузере. **Html-теги** или **html-элементы** – это специальные дескрипторы, из которых состоит html-документ.

Html-документ иначе еще называют html-страницей или веб-страницей. Html-документы – это текстовые файлы с расширением .html или .htm, содержимое которых соответствует спецификации HTML.

Веб-сайт – это совокупность html-документов, взаимосвязанных темой, расположением и владельцем.

1.2.3. Валидация HTML кода

Если в HTML документе есть ошибки, то он все равно будет отображаться в браузере. Это происходит потому, что все браузеры имеют встроенные алгоритмы исправления ошибок и дополнения недостающего кода. Если в html-документе есть ошибки, то в любом случае, браузер будет пытаться отобразить документ, руководствуясь своими алгоритмами. Но в результате страница может отобразиться не так, как этого хотел разработчик. Корректность написания htm-кода нужно проверять. Если код небольшой, то можно просто просмотреть весь код вручную и найти ошибки. Некоторые редакторы кода, например, Adobe Dreamweaver выводят сообщения об ошибках. Но лучше всего проверять страницы в сервисе валидации разметки https://validator.w3.org/.

Его создал и поддерживает W3C. Сервис проверяет html-код и составляет отчет по ошибкам в нем. Можно проверить файл целиком или скопировать и ввести часть кода.

Для проверки кода надо:

1. Открыть сервис валидации разметки в браузере.
2. Перейти на вкладку Validate by Direct Input.
3. Скопировать весь код документа (не только body) и вставьте в место для ввода.
4. Нажать на Check (проверить).

1.2.4. Структура HTML документа

Если разместить на странице контент без разбивки на семантические элементы, то это может иметь негативные последствия.

1. Поисковые системы, индексирующие страницу, считают содержание заголовков важными ключевыми словами для влияния на ранжирование поиска страницы. Без заголовков страница будет плохо работать с точки зрения SEO.
2. Пользователи, просматривающие веб-страницу, быстро сканируют ее в поиске подходящего контента, часто просто просматривая только заголовки. Если они не смогут увидеть ничего полезного в течение нескольких секунд, они, скорее всего, уйдут со страницы.
3. Сильно слабовидящие люди часто не читают веб-страницы, а слушают их. Это делается с помощью программы чтения с экрана. Это программное обеспечение предоставляет способы быстрого доступа к текстовому

контенту. Среди различных используемых методов программы-ридеры предоставляют схему документа, считывая заголовки и позволяя своим пользователям быстро находить нужную им информацию. Если заголовки недоступны, пользователи будут вынуждены прослушивать весь документ.

4. Чтобы стилизовать контент с помощью CSS или сделать его интересным с помощью JavaScript, нужно, чтобы элементы обертывали соответствующий контент, чтобы CSS и JavaScript смогли эффективно работать.

HTML-документ формально делится на три части. В первой, не обязательной части, содержится информация об используемой версии языка HTML и правилах его форматирования. Вторая и третья части должны быть обязательно заключены в тег <html>, т.е. сам HTML – документ начинается с открывающего тега <html> и заканчивается закрывающим тегом </html>. Во второй части размещается заголовок документа, а в третьей – его тело. Html-документ обязательно должен содержать теги <html>, <head>, <body>. На рисунке 7 приведен пример простейшего HTML-документа

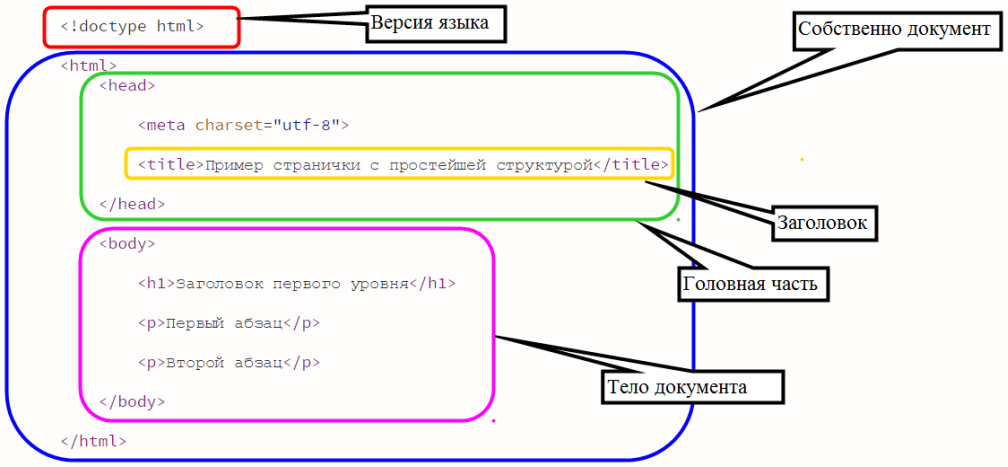


Рисунок 7 – Части веб- документа

Элемент <!doctype html> объявляет тип документа.

Элемент <html> иногда называют «корневым элементом». Он является контейнером для всех других элементов, за исключением тега <!doctype html>. Для упрощения работы поисковых систем и браузеров в тег <html> включают атрибут lang, который содержит информацию о языке веб-страницы.

Элемент <head> выступает в качестве контейнера для всего содержимого, которое необходимо включить в HTML документ, но нет надобности показывать посетителям страницы. Он включает такие вещи, как ключевые слова и описание страницы для поисковых машин, CSS для стилизирования контента, объявление поддерживаемого набора символов и многое другое.

<meta charset="utf-8"> устанавливает в качестве символьной кодировки для документа utf-8, который включает большинство символов из множества языков. По существу, страница с такой кодировкой сможет отобразить любой текстовый контент.

Элемент <title> устанавливает заголовок страницы, который появляется во вкладке браузера, загружающей эту страницу, также это заглавие используется при описании страницы, когда ее сохраняют в закладках или избранном.

Элемент <body> содержит весь контент, который будет показан пользователям.

1.2.5. Секция заголовка <head>

Секция заголовка <head> может содержать служебную информацию для описания и индексирования документа поисковыми машинами, подключения ресурсов из внешних источников, скрипты, стили. Теги секции <head> не отображается веб-обозревателями, исключение составляет <title>.

Внутри контейнера <head> допускается размещать следующие элементы:

<title> – определяет заголовок вкладки браузера в которой открыта страница;

<meta> – содержит информацию для браузера и поисковых машин;

<style> – содержит внутренние CSS-стили для элементов страницы;

<link> – подключает внешние CSS-стили (.css) и другие ресурсы;

<script> – является контейнером для javaScript-кода и подключает внешние файлы со скриптами (.js);

<noscript> – показывает свое содержимое, если браузер не поддерживает работу со скриптами или их поддержка отключена пользователем;

<base> – определяет URL-базу для всех относительных URL на странице и значение target по умолчанию для гиперссылок.

Ниже на рисунке 8 приведен пример типичной секции head.

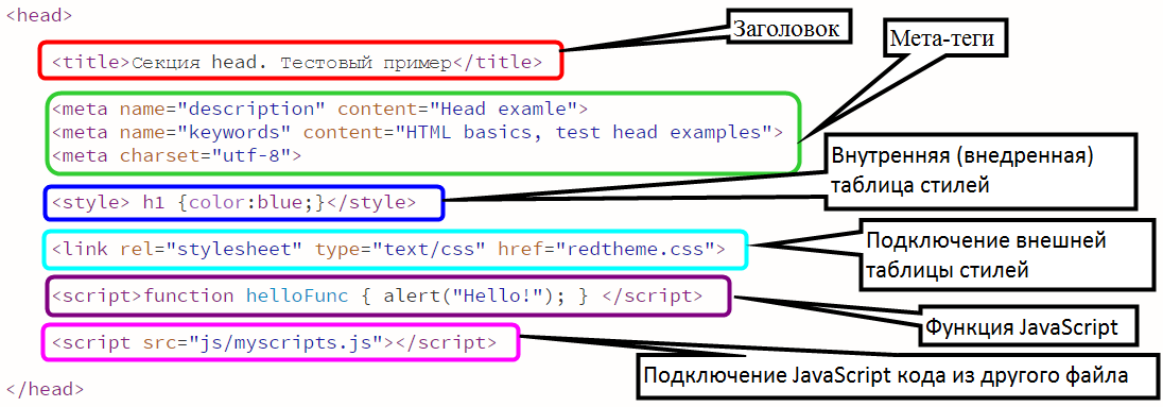


Рисунок 8 – секция head

<title>

Элемент <title> может отображаться только как название окна в браузере. В документе может быть только один тег <title>. Если в документе отсутствует тег <title>, то документ не будет считаться валидным.

Элемент <title>:

определяет заголовок на панели инструментов браузера;

даёт название для страницы, когда она добавляется в Избранное;

отображает заголовок страницы в результатах поисковой системы.

Тег <title> используется поисковыми машинами для идентификации содержимого документа и поэтому необходимо давать осмысленные заголовки. Так, вместо такого заголовка, как «Введение», который не даёт достаточно информации о документе, надо записать, например, «Введение в высшую математику». В идеале в заголовок должны войти все ключевые слова по которым предполагается продвигать страницу в поисковых системах, а лучше целый запрос. Название должно быть не более 70 символов. Поисковые машины не воспринимают более длинный текст. Кроме того, название, превышающее этот размер, может не поместиться в строку заголовка окна браузера.

```
<title>Секция head. Тестовый пример</title>
```

Если страница создаётся в какой-либо системе управления контентом, то CMS часто автоматически вносит стандартный ***<title>***. Если забыть его переопределить, то может случиться, что страница будет продвигаться в поисковой машине по такому, автоматически созданному, заголовку. Пример неудачного продвижения сайтов, которые продвигаются по фразе «Документ без названия» на рисунке 8.

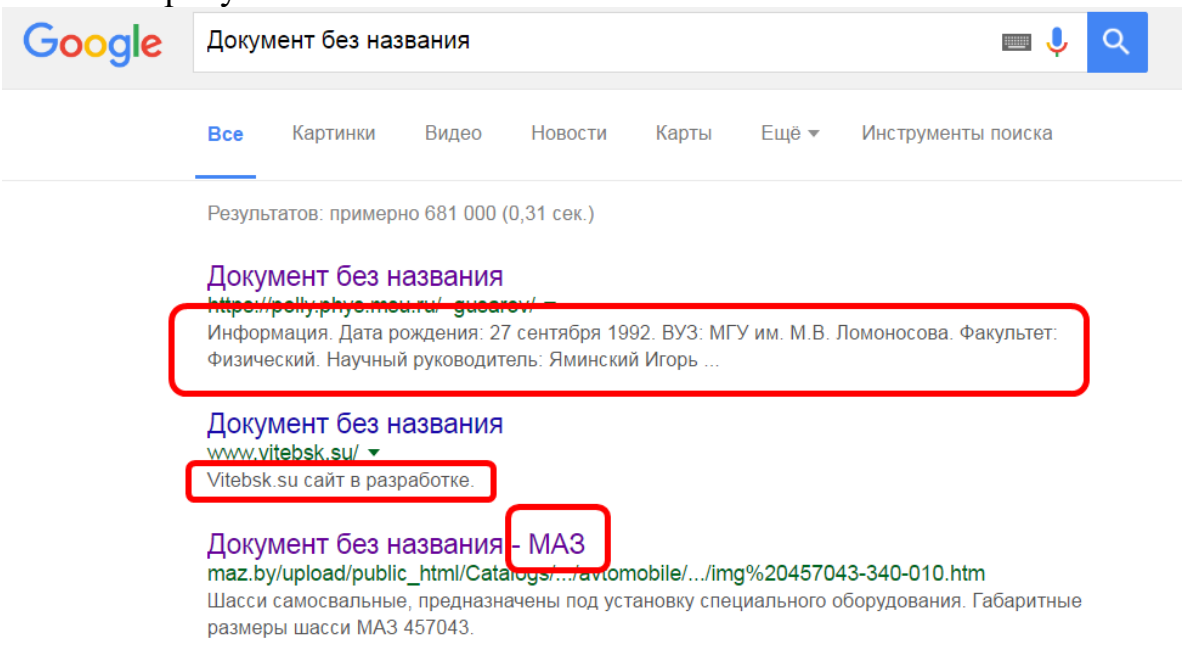


Рисунок 9 – пример запроса

<meta>

Мета-теги содержат служебную информацию, не отображаемую в окне браузера, или метаданные. Метаданные – это информация о данных, в данном случае о документе HTML. Метаданные не отображаются на странице, но обрабатываются поисковой машиной (ключевые слова) и используются браузером (информация как отображать страницу) или другими веб-службами.

Мета-элементы обычно используются для указания описания страниц, автора документа, последнего изменения и т.д. В частности метаданные могут представлять собой собственные форматы социальных сетей или других сайтов, созданные для предоставления им специальной информации, которую они могут использовать.

Разрешается использовать более чем один мета-тег, но все они должны быть размещены в контейнере <head>.

Пример, определения описания страницы:

```
<meta name="description" content="Head example">
```

Пример, определение автора страницы:

```
<meta name="author" content="Ivanov Ivan">
```

Пример, обновление документа каждые 30 секунд:

```
<meta http-equiv="refresh" content="30">
```

У тега <meta> есть несколько атрибутов.

name

Атрибут name указывает имя для информации, которая хранится в атрибуте content. Часто используется для указания имени автора. В этом случае атрибуту name присваивается значение author, и поисковые системы смогут найти нужный сайт по имени автора.

```
<meta name="author" content="Ivanov, Иванов Иван">
```

Может содержать краткое описание сайта, используемое поисковой машиной для индексирования. В этом случае атрибуту name присваивается значение description.

```
<meta name="description" content="The best homepage">
```

Часто description используют, чтобы повлиять на выбор сниппета поисковой машиной. Согласно результатам исследования американского SEO-специалиста Эвана Холла на первой странице выдачи Google выводит метатег description для сниппета примерно в 30% случаев.

Мета-тег может содержать ключевые слова, используемые поисковым роботом. В этом случае атрибуту name присваивается значение keywords. Длина содержимого мета-тегов keywords не должна превышать 1000 символов. В настоящее время в плане продвижения практически не даёт эффекта, но слова, входящие в content включаются в общее число при подсчете повторений слов на странице.

```
<meta name="keywords" content="META, HTML, WWW, Web, паутина, поиск,
определение, рекомендации, примеры использования, учебник,
руководство, справка">
```

Мета тег может указывать на один из пакетов программного обеспечения, используемых для создания документа (не используется на страницах, созданных вручную). В этом случае атрибуту name присваивается значение generator.

```
<meta name="generator" content="Joomla! 1.5 - Open Source Content
Management">
```

В HTML5 был введен метод, позволяющий контролировать видимую область на веб-странице с помощью тега <meta>. В этом случае name устанавливается значение viewport. Ниже приведен пример настройки области просмотра, чтобы сайт выглядел хорошо на всех устройствах.

```
<meta name="viewport" content="width=device-width, initial-
scale=1.0"/>
```

http-equiv

Атрибут http-equiv служит в качестве альтернативы атрибуту name. Используется так же, как и атрибут name. Атрибут http-equiv может использоваться для имитации заголовка ответа HTTP. При отображении страницы браузер будет следовать инструкциям, заданным в атрибуте, при отсутствии их в ответе сервера.

Может принимать значения content-type, default-style или refresh.

Например, content-type указывает тип документа:

```
<meta http-equiv = "content-type" content = "text/html; charset = UTF-8">
```

Атрибут http-equiv, определенное как default-style, указывает на таблицу стилей по умолчанию.

```
<meta http-equiv="default-style" content="the document's preferred stylesheet ">
```

Атрибут http-equiv может использоваться для автоматического обновления html-документа через определенное количество секунд. В этом случае атрибуту http-equiv присваивается значение refresh.

Пример перезагрузки html-документа через каждые 7 секунд:

```
<META http-equiv="refresh" content="7">
```

Пример перенаправления пользователя по истечении 7 секунд на сайт bsu.by.

```
<meta http-equiv="refresh" content="7";
url="http://www.bsu.by>"
```

content

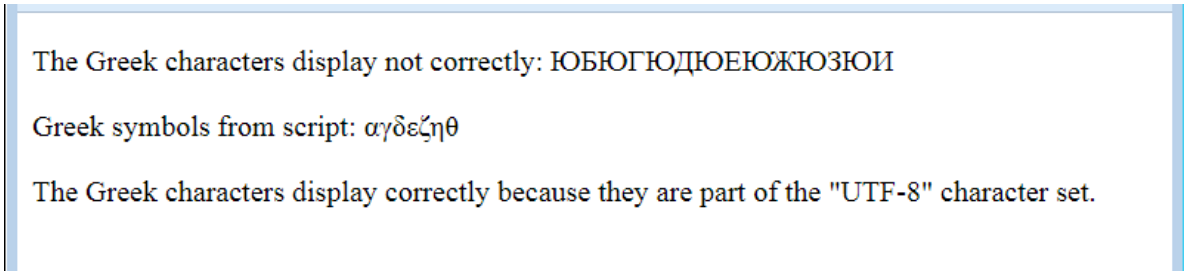
Атрибут content указывает значение, связанное с атрибутом http-equiv или name.

charset

Атрибут определяет кодировку символов, которая должна быть использована на компьютере пользователя при просмотре данного html-документа.

```
<meta charset="utf-8">
<!doctype html>
<html>
  <body>
    <p>The Greek characters display not correctly: αγδεζηθ</p>
    <script charset="UTF-8" src="demoScriptCharset.js">
    </script>
    <p>The Greek characters display correctly because they are part of the "UTF-8" character set.</p>
  </body>
</html>
Текст файла demoScriptCharset.js:
document.write("Greek symbols from script: αγδεζηθ");
```

Результат выполнения:



Как видно из результата выполнения, в первом абзаце греческие символы отображаются не правильно, потому что в документе не прописан мета-тег для указания кодировки `utf-8 <meta charset="utf-8">`, а греческие символы являются частью набора символов «UTF-8».

При работе скрипта греческие символы отображаются правильно, потому что в теге <script> задаётся кодировка символов «UTF-8» через атрибут charset="UTF-8".

<style>

Внутренние (внедренные) таблицы стилей CSS расположены внутри html-документа, в теге <style>. Каждый html -документ может содержать несколько тегов <style>.

<link>

Тег устанавливает связь с внешними ресурсами, например, присоединяет стилевые файлы, файлы шрифтов, иконки и т.д. Элемент <link> не содержит контента, тег содержит только атрибуты. Синтаксис тега, следующий:

```
<head>
<link атрибуты>
</head>
```

Тег имеет следующие атрибуты:

href содержит путь к связываемому файлу;

hreflang указывает язык текста в связанном документе;

media определяет устройство, на каком будет отображаться связанный документ;

rel определяет отношения между текущим документом и файлом, на который делается ссылка;

sizes указывает размер иконок для визуального отображения;

type содержит MIME-тип данных подключаемого файла.

Примеры:

```
<link rel="stylesheet" href="ie.css">
<link rel="shortcut icon" href="http://mysite.by/apple.ico">
```

Тег <link> также может содержать глобальные атрибуты.

<base>

Элемент <base> определен внутри контейнера <head> и указывает базовый URL для всех относительных URL-адресов в документе. В документе может быть только один элемент <base>.

Атрибуты тега <base>

href

Тег <base> предназначен для документов, в которых используется относительный адрес и эти документы могут переноситься в другую папку или даже на другой компьютер без потери связи. Браузер ищет тег <base>, определяет полный адрес документа и корректно загружает его. Например, если адрес документа указан как <base href="http://www.megasite.by/news/">, то при добавлении рисунков достаточно использовать относительный адрес <img src="images/elephant.gif">. При этом полный путь к изображению будет http://www.megasite.by/news/images/elephant.gif, что позволяет браузеру всегда находить графический файл, независимо от того, где находится текущая веб-страница.

target

В теге <base> можно указать имя окна или фрейма, куда будет загружаться документ, открываемый по ссылке. В зависимости от значения атрибута документ может быть открыт:

_blank – в новом окне или вкладке;

_self – в том же окне (значение по умолчанию);

_parent – в родительском окне;

_top – в полноэкранном окне;

framename – в именованном фрейме.

Пример, ссылка откроется в новой вкладке.

```
<!doctype html>
<html>
  <head>
    <base target="_blank">
    <meta charset="utf-8">
  </head>
  <body>
    <a href="Examples for presentation html5 1 a.htm">Новый
документ</a>
  </body>
</html>
```

1.2.6. Комментарии в html-документе. <!--…-->

В HTML есть возможность писать комментарии в коде. Комментарии игнорируются обозревателем и не видны пользователю, их добавляют для того, чтобы пояснить, как работает написанный код, что делают отдельные его части и т. д. Комментарии задаются между маркерами <!-- и -->.

Пример использования комментариев.

```
<html><!--Начало документа-->
  <head>
    <meta charset="utf-8">
    <title> Пример использования комментариев </title>
  </head>
  <body><!--Начало тела документа-->
    <!--Внутри тега комментарий поместить нельзя-->
```

```
    <!--А собственно текста в документе нет-->
  </body><!--Конец тела документа-->
</html><!--Конец документа-->
```

1.2.7. Атрибуты событий

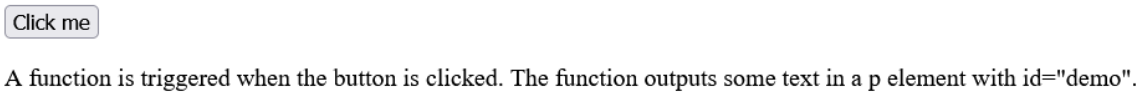
В HTML есть возможность запускать различные действия в браузере при наступлении определенных событий, например, запускать JavaScript код, когда пользователь нажимает на кнопку.

Глобальных атрибутов событий или событийных атрибутов, которые можно добавлять к элементам HTML для определения наступления событий довольно много, их можно поделить на несколько групп.

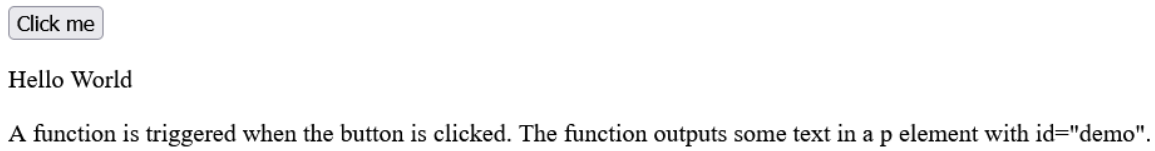
Ниже приведен пример шаблона для обработки события (в данном случае ***onclick***). В значении атрибута стоит вызов функции javascript, а может стоять любой фрагмент javascript кода.

```
<!doctype html>
<html>
  <body>
    <button onclick="myFunction()">Click me</button>
    <p id="demo"></p>
    <p>A function is triggered when the button is clicked. The
function outputs some text in a p element with id="demo".</p>
    <script>
      function myFunction() {
        document.getElementById("demo").innerHTML = "Hello World";
      }
    </script>
  </body>
</html>
```

Первоначально абзац с атрибутом id равным demo не имеет текста.



При нажатии на кнопку в абзац с атрибутом id равным demo добавляется текст.



Подробно об атрибутах событий можно почитать в HTML справочнике https://www.w3schools.com/tags/ref_eventattributes.asp.

Атрибуты событий окна

Перечислим атрибуты событий окна.

onafterprint – скрипт запускается после печати документа;

onbeforeprint – скрипт запускается до того, как документ будет напечатан;

onbeforeunload – скрипт запускается, когда документ будет выгружен;

onerror – скрипт запускается при возникновении ошибки;

onhashchange скрипт запускается, когда произошли изменения в якорной части URL-адреса

onload скрипт запускается, когда страница полностью загружена, включая содержание, изображения, стилевые файлы и внешние скрипты;

onmessage – скрипт запускается при запуске сообщения;

onoffline – скрипт запускается, когда браузер начинает работать в автономном режиме;

ononline – скрипт запускается, когда браузер начинает работать в Интернете;

onpagehide – сценарий запускается, когда пользователь уходит со страницы;

onpageshow – сценарий запускается, когда пользователь переходит на страницу;

onpopstate – сценарий запускается при изменении истории окна;

onresize – сценарий запускается при изменении размера окна браузера;

onstorage – сценарий запускается при обновлении области веб-хранилища;

onunload – сценарий запускается, когда страница выгружается (или окно браузера закрывается).

Атрибуты событий формы

onblur – сценарий запускается, когда элемент теряет фокус;

onchange – сценарий запускается, когда изменяется значение элемента;

oncontextmenu – сценарий запускается при вызове контекстного меню;

onfocus – сценарий запускается, когда элемент получает фокус;

oninput – сценарий запускается, когда пользователь начинает ввод в элемент;

oninvalid – сценарий запускается, если значение элемента не валидно;

onreset – сценарий запускается при нажатии кнопки Reset в форме;

onsearch – сценарий запускается, когда пользователь что-то пишет в поле поиска (для <input = "search">);

onselect – сценарий запускается после того, как какой-либо текст был выбран в элементе;

onsubmit – сценарий запускается при отправке формы.

Атрибуты событий клавиатуры

onkeydown – сценарий запускается, когда пользователь нажимает клавишу;

onkeypress – сценарий запускается, когда пользователь нажимает клавишу;

onkeyup – сценарий запускается, когда пользователь отпускает клавишу.

Атрибуты событий мыши

onclick – сценарий запускается, когда на элементе происходит щелчок мыши;

ondblclick – сценарий запускается, когда на элементе происходит двойной щелчок мыши;

onmousedown – сценарий запускается при нажатии кнопки мыши на элементе;

onmousemove – сценарий запускается, когда указатель мыши движется над элементом;

onmouseout – сценарий запускается, когда указатель мыши перемещается за границы элемента;

onmouseover – сценарий запускается, когда указатель мыши перемещается над элементом;

onmouseup – сценарий запускается, когда кнопка мыши отпускается над элементом;

onwheel – сценарий запускается, когда колесико мыши прокручивается над элементом.

Атрибуты событий при перетаскивании

ondrag – сценарий запускается, когда курсор двигается при перетаскивании;

ondragend – сценарий запускается, когда пользователь отпускает курсор мыши в процессе перетаскивания;

ondragenter – сценарий запускается, когда перетаскиваемый элемент достигает конечного элемента;

ondragleave – сценарий запускается, когда курсор мыши покидает пределы перетаскиваемого элемента;

ondragover – сценарий запускается, когда курсор мыши наведен на элемент при перетаскивании;

ondrop – сценарий запускается, когда происходит drop (бросание) элемента;

dragstart – сценарий запускается, когда пользователь начинает перетаскивание элемента.

Атрибуты событий в буфере обмена

oncopy – сценарий запускается, когда пользователь копирует содержимое элемента;

oncut – сценарий запускается, когда пользователь вырезает содержимое элемента;

onpaste – сценарий запускается, когда пользователь вставляет некоторый контент из буфера обмена в элемент.

Атрибуты медиа-событий

onabort – сценарий запускается при прерывании;

oncanplay – сценарий запускается, когда файл достаточно буферизирован и готов начать воспроизведение;

oncanplaythrough – сценарий запускается, когда файл может быть воспроизведен до конца без паузы для буферизации;

oncuechange – сценарий запускается, когда происходит изменение метки в элементе <track>;

ondurationchange – сценарий запускается при изменении длины носителя;

onemptied – сценарий запускается, когда файл становится недоступен;

onended – сценарий запускается, когда медиа достигает конца (полезное событие для таких сообщений, как «спасибо за прослушивание»);

onerror – сценарий запускается, когда возникает ошибка при загрузке файла;

onloadeddata – сценарий запускается при загрузке медиа данных;

onloadedmetadata – сценарий запускается, когда загружаются метаданные (например, размеры и продолжительность);

onloadstart – сценарий запускается, когда файл начинает загружаться;

onpause – сценарий запускается, когда воспроизведение приостанавливается либо пользователем, либо программно;

onplay – сценарий запускается, когда носитель готов начать воспроизведение;

onplaying- сценарий запускается при начале воспроизведения медиа;

onprogress – сценарий запускается, когда браузер находится в процессе получения медиа данных;

onratechange – сценарий запускается каждый раз при изменении скорости воспроизведения (например, когда пользователь переключается в режим замедленного воспроизведения или быстрого перехода вперед);

onseeked – сценарий запускается, когда атрибут seek установлен равным false, что указывает на то, что поиск закончился;

onseeking – сценарий запускается, когда атрибут seek установлен равным true что указывает на то, что поиск активен;

onstalled – сценарий запускается, когда браузер не сможет получить данные мультимедиа по любой причине;

onsuspend – сценарий запускается, когда при извлечении медиа данных происходит остановка до того, как они будут полностью загружены по любой причине;

ontimeupdate – сценарий запускается, когда позиция проигрывания изменилась (например, когда пользователь быстро перейдет в другую точку);

onvolumechange – сценарий запускается при изменении громкости;

onwaiting сценарий запускается, когда носитель приостановлен, но ожидается его возобновление.

1.2.8. Прочие атрибуты

onshow – сценарий запускается, когда элемент <menu> отображается как контекстное меню;

ontoggle – сценарий запускается, когда пользователь открывает или закрывает элемент <details>.

1.2.9. Специальные символы и диакритические знаки

Некоторые символы, которые иногда нужно поместить на страницу, отсутствуют на обычной клавиатуре. Например, математические, технические, валютные символы или символы национальных алфавитов, которые не входят в

базовую часть таблицы ASCII кодов. Некоторые символы присутствуют на клавиатуре, но их нельзя непосредственно ввести в html-документ. Например, символы меньше и больше зарезервированы для обозначения тегов. Такие символы можно вывести на странице при помощи специальных кодов, каждый из которых начинается со знака **&**, за которым следует название символа либо его числовой код в десятичной или шестнадцатеричной системе. Завершается код символом точка с запятой. Ниже в таблице 4 приведены некоторые специальные символы.

Таблица 4. Специальные символы

Имя	Код	Вид	Описание
&quot;	&#34;	"	двойная кавычка
&amp;	&#38;	&	амперсанд
&lt;	&#60;	<	знак 'меньше'
&gt;	&#62;	>	знак 'больше'
			неразрывный пробел
&sect;	&#167;	§	параграф
&copy;	&#169;	©	знак copyright
&laquo;	&#171;	«	левая двойная угловая скобка
&reg;	&#174;	®	знак зарегистрированной торговой марки
&deg;	&#176;	°	градус
&plusmn;	&#177;	±	плюс-минус
&acute;	&#180;	´	знак ударения
&middot;	&#183;	·	точка
&raquo;	&#187;	»	правая двойная угловая скобка
&ndash;	&#8211;	–	тире
&mdash;	&#8212;	—	длинное тире
&lsquo;	&#8216;	‘	левая одиночная кавычка
&rsquo;	&#8217;	’	правая одиночная кавычка
&sbquo;	&#8218;	‚	нижняя одиночная кавычка
&ldquo;	&#8220;	“	левая двойная кавычка
&rdquo;	&#8221;	”	правая двойная кавычка
&bdquo;	&#8222;	„	нижняя двойная кавычка
&frasl;	&#8260;	⁄	косая дробная черта
&euro;	&#8364;	€	евро
	&#8470;	№	знак номера
&trade;	&#8482;	™	знак торговой марки
&loz;	&#9674	◊	ромб
	&#9675	○	круг

1.2.10. Формы

Практически на каждом сайте есть поля для ввода текста, кнопки, переключатели, выпадающие списки или флажки.

Форма предназначена для обмена данными между пользователем и сервером. Область применения форм не ограничена отправкой данных на сервер, с помощью клиентских скриптов можно получить доступ к любому элементу формы, изменять его и применять по своему усмотрению.

Документ может содержать любое количество форм, но одновременно на сервер может быть отправлена только одна форма. По этой причине данные форм должны быть независимы и не могут вкладываться друг в друга.

Нельзя размещать форму внутри другой формы. Это может привести к непредсказуемому поведению форм, в зависимости от браузера.

Тег <form> определяет форму на веб-странице и служит контейнером для остальных элементов интерфейса. Внутри тега <form> помещаются теги, которые определяют элементы для ввода информации: текстовые поля, радиокнопки, флажки и т.д. Сама форма никак не отображается на веб-странице, видны только ее элементы и результаты вложенных тегов.

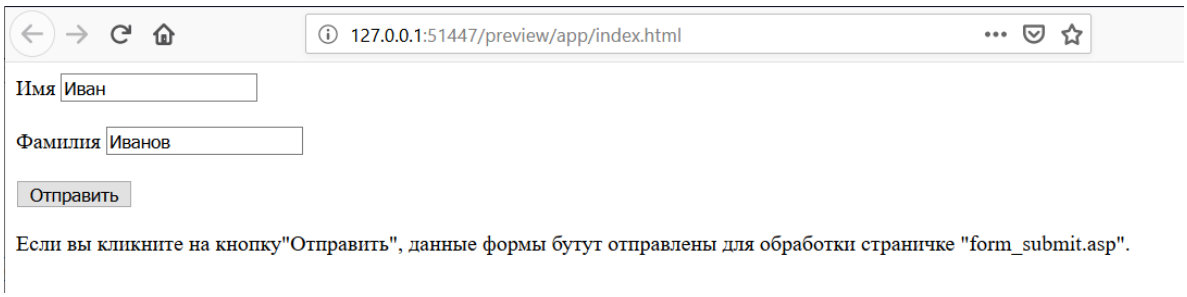
Тег <form> может содержать следующие элементы формы:

- <input> наиболее часто используемый элемент, может отображаться несколькими способами, в зависимости от атрибута type;
- <label> метка для элементов формы;
- <select> выпадающий список;
- <textarea> многострочная текстовая область для ввода;
- <button> кнопка;
- <fieldset> используется для выделения группы элементов;
- <legend> определяет заголовок для <fieldset> элемента;
- <datalist> список предопределенных вариантов;
- <output> результат вычисления;
- <option> раскрывающий список;
- <optgroup> определяет группу в выпадающем списке.

Приведем пример простейшей формы.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Простейшая форма</title>
  </head>
  <body>
    <form name="nameForm" action="form_submit.asp">
    Имя <input type="text" name="fname" value="Иван"><br><br>
    Фамилия <input type="text" name="lname" value="Иванов"><br><br>
    <input type="submit" value="Отправить">
    </form>
    <p>Если вы кликните на кнопку "Отправить", данные формы будут
отправлены для обработки страничке "form_submit.asp".</p>
  </body>
```

```
</html>
```



В данном случае на форме есть два текстовых поля для ввода. Кнопка для отправки данных на сервер определяет тег <input> с атрибутом `type="submit"`. Данные формы отправляются на веб-страницу form_submit.asp на сервере.

Перед отправкой данных браузер представляет информацию в виде пар «имя=значение», где имя определяется атрибутом name элемента формы, а значение – введенными пользователем данными. Стандарт URL позволяет использовать в ссылках только ограниченный набор символов: латинские буквы, цифры и лишь некоторые знаки препинания. Если нужно использовать в URL символы кириллицы, или иероглифы, или, скажем, специфические символы французского языка, то не входящие в стандарт символы должны быть перекодированы особым образом. В Интернете достаточно много сервисов для кодировки и раскодировки url-строк. Например, http://www.codenet.ru/services/urlencode-urldecode/.

Если из формы методом GET передаётся строковая переменная, содержащая текст на русском языке, то при передаче из формы переменная приходит в кодировке utf-8. Например, если в примере, описанном выше, передаются значения по умолчанию Иван Иванов, то адресная строка будет иметь вид:

file:///C:/Lena/html5/2016/form/form_submit.asp?fname=%D0%98%D0%B2%D0%B0%D0%BD&lname=%D0%98%D0%B2%D0%B0%D0%BD%D0%BE%D0%B2

где %D0%98%D0%B2%D0%B0%D0%BD соответствует значению Иван, а %D0%98%D0%B2%D0%B0%D0%BD%D0%BE%D0%B2 – Иванов.

После нажатия пользователем кнопки submit, происходит вызов обработчика формы, который задаётся атрибутом action тега <form>. Обработчиком формы обычно выступает программа, написанная на любом серверном языке, например, PHP, Python, C# и т.д. Если на форме нет кнопки Submit, то данные на сервер можно отправить, нажав клавишу Enter.

Если данные не отправляются на сервер, а обрабатываются, например, java скриптом, то вместо кнопки типа "submit" можно воспользоваться элементом <input> с атрибутом `type = "button"`.

Атрибуты тега <form>

Все атрибуты являются опциональными, но принято указывать атрибуты action и method.

Атрибут action определяет url программы-обработчика формы, т.е. куда адресуется http-запрос, по умолчанию – это адрес самой html-страницы, где

расположена форма. Если значение атрибута не будет указано, то после перезагрузки страницы элементы формы примут значения по умолчанию. В случае, если вся работа будет выполняться на стороне клиента сценариями JavaScript, то для атрибута action можно указать значение #. Также можно сделать так, чтобы заполненная посетителем форма приходила на электронную почту:

```
 <form action="mailto:адрес электронной почты"
enctype="text/plain"></form>
```

Атрибут method – метод http-запроса, по умолчанию – GET. Если для отправки данных используется метод GET, то адресная строка, после имени страницы может стоять вопросительный знак, после которого перечисляются параметры. Параметры разделяются между собой символом амперсанда (&). Нелатинские символы преобразуются в шестнадцатеричное представление (в форме %HH, где HH – шестнадцатеричный код для значения ASCII-символа), пробел заменяется на плюс (+). Например, mysite.com/handler.php?login=Anna&pass=123456. Как видно, метод get не является безопасным для пересылки конфиденциальных данных. Кроме GET и POST существуют и другие методы, например, PUT, PATCH и DELETE. Разработчики выбирают тот или иной метод исходя из смысла действия.

Атрибут name содержит имя формы, используемое для ее идентификации, например, в DOM. Атрибут необходим, если на странице расположено несколько форм. Этот атрибут был введен для обеспечения обратной совместимости, теперь активно используется атрибут id для идентификации элементов. В целях совместимости лучше инициализировать оба атрибута, и name, и id, одинаковым значением.

Атрибут accept-charset содержит список кодировок, необходимых серверу для обработки данных, пересылаемых формой. Значением является список кодировок, разделенных пробелами или запятыми, по умолчанию кодировка совпадает с кодировкой страницы.

Атрибут autocomplete – браузер выводит подсказки на основе значений, которые пользователь ввел ранее, если автозаполнение включено (рис. 10).

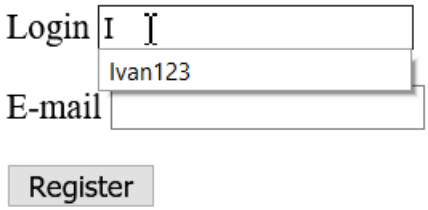


Рисунок 10 – работа автозаполнения

Атрибут novalidate используется, если браузер не должен проверять форму.

Атрибут target задаёт имя окна или фрейма, куда обработчик будет загружать возвращаемый результат.

Атрибут enctype указывает, каким образом будут закодированы данные формы при отправке на сервер. Применяется только для метода POST. Может принимать значения application/x-www-form-urlencoded, multipart/form-data, text/plain. application / x-www-form-urlencoded является значением по умолчанию. Все символы кодируются перед отправкой: пробелы преобразуются в

символы «+», а специальные символы преобразуются в значения ASCII HEX. При значении multipart/form-data никакие символы не кодируются. Это значение используется в формах, у которых есть контроль загрузки файлов. При значении text/plain пробелы преобразуются в символы «+», но специальные символы не кодируются.

Атрибут rel определяет связь между связанным ресурсом и текущим документом.

Подпись к элементу пользовательского интерфейса <label>

Тег <label> используется для создания поясняющих текстов или меток к элементам управления html-форм. Он устанавливает связь между определенной текстовой меткой и элементом формы <input>. При установлении связи при нажатии курсором мыши на тексте подсказки связанный с ней элемент формы получает фокус. Это полезно, когда элемент занимает достаточно маленькую область экрана, и пользователям трудно нажимать на эти небольшие области, например флажок. При использовании меток, когда пользователь щелкает текст внутри <label> элемента, он переключает ввод на нужный элемент и тем самым увеличивается область нажатия. Программы чтения с экрана, скринридеры, будет проговаривать вслух содержимое метки, когда связанный с ней элемент будет находиться в фокусе.

Для элемента может быть создано несколько меток. Кроме того, с помощью <label> можно устанавливать горячие клавиши на клавиатуре и переходить на активный элемент подобно ссылкам.

Метки могут использоваться не только для элементов форм.

Существует два способа связывания объекта и метки. Первый заключается в использовании идентификатора id внутри тега <input> и указании его имени в качестве параметра for тега <label>. При втором способе тег <input> помещается внутрь контейнера <label>. В браузерах применение <label> ничем не отличается от надписей, сформированным обычным текстом.

Атрибуты тега <label>

Тег <label> имеет два атрибута: for и form.

Атрибут for содержит идентификатор элемента html-формы, с которым связывается текущая метка. Атрибут form содержит идентификатор формы, с которой связывается метка.

Приведем пример использования меток к элементам формы.

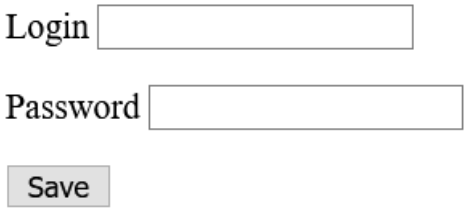


Рисунок 11 – форма ввода пароля

```
<!doctype html>
<html>
<head>
```

```
 <meta charset="utf-8">
 <title>Использование надписи label в форме</title>
 </head>

 <body>
  <form id="frm1" name="frm1" action="hand.php" method="post"
enctype="application/x-www-form-urlencoded" >
   <p>
    <label for="login" accesskey="l">Login</label>
    <input id="login" name="login" value="">
   </p>
   <p>
    <label for="psw">Password</label>
    <input type="password" id="psw" name="psw" value="">
   </p>
   <p>
    <input type="submit" value="Save">
   </p>
  </form>
 </body>
 </html>
```

1.2.11. Общие атрибуты для элементов формы

Многие теги, используемые для определения элементов формы, имеют собственные атрибуты, но существует набор общих атрибутов, присущих всем элементам формы.

autofocus

Атрибут autofocus логического типа указывает, что поле ввода должно автоматически получать фокус при загрузке страницы. Атрибут явно определяется только для одного элемента в форме.

disabled

Атрибут логического типа определяет, может ли пользователь взаимодействовать с элементом. Если атрибут не определен, то по умолчанию пользователь может взаимодействовать с элементом.

form

Не все элементы формы должны размещаться в форме. Теоретически возможно помещать элементы формы за рамками элемента <form>, при этом атрибут содержит id формы в том же документе, с которой он ассоциирован.

name

Содержит имя элемента. Параметр name элементов формы использует как браузер, так и сервер: браузер узнает, какие имена дать каждой части данных, а сервер может получить эти данные, обратясь к ним по заданному имени. Данные формы отправляются на сервер в виде пары имя/значение. Для полей формы важно определить атрибут name, т.к. в http-запрос попадают только те зна-

чения, для которых в поле задан name. Значение для атрибута name обычно уникально в пределах формы. Например, дана форма

```
<form action="foo.php" method="get">
  <div>
    <label for="contact">Как вы хотите, чтобы к вам
обращались?</label>
    <input name="contact" id="contact" value="Лапушка">
  </div>
  <div>
    <label for="check">Вы хотите получать напоминания?</label>
    <input     type="checkbox" name="check" id="check" value="Да">
  </div>
  <div>
    <button>Запомнить мои предпочтения</button>
  </div>
</form>
```

value

Содержит значение параметра, которое будет передано на сервер. Имя этого параметра хранится в name.

Просмотреть HTTP-запрос можно, например, в Инструментах разработчика Firefox (рисунок 12). Они открываются при нажатии клавиши F12 в браузере Firefox. Надо выбрать вкладку Сеть, затем вкладку Заголовки и выбрать файл, указанный в атрибуте action формы. В данном случае это foo.php.

```
contact=Лапушка&check=Да
Реально параметры равны
contact=%D0%9B%D0%B0%D0%BF%D1%83%D1%88%D0%BA%D0%B0
check=%D0%94%D0%B0
ниже.
```

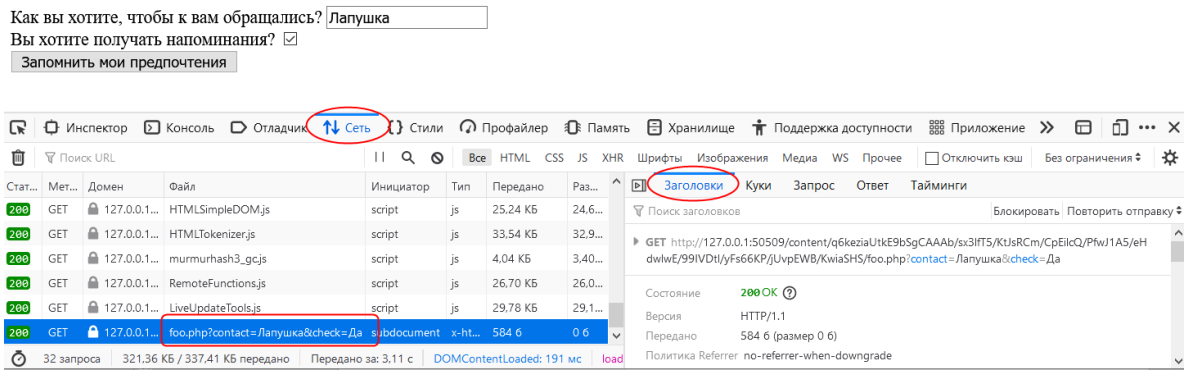


Рисунок 12 – HTTP-запрос в Инструментах разработчика Firefox.

1.2.12. Элементы формы. Тег <input>

Для построения различных элементов формы можно использовать тег <input>. С помощью этого тега можно создавать текстовые поля, различные кнопки, переключатели, флажки и т. д. Тег <input> не обязательно размещать внутри контейнера <form>, определяющего форму, но, если введенные пользователем данные должны быть отправлены на сервер или обработаны с помощью скриптов на языке JavaScript, то <input> обязательно помещать внутрь

контейнера <form>. Тег <input> может отображаться несколькими способами, в зависимости от атрибута type.

Атрибут type

Атрибут type определяет внешний вид и поведение элемента <input>. Он может принимать следующие значения:

button – кнопка;

checkbox – флажок;

color – определяет интерфейс для выбора цвета;

date – интерфейс для выбора календарной даты;

datetime-local указание местной даты и времени;

email – текстовое поле для адреса электронной почты, проверка ошибок на стороне клиента, выполняемая браузером;

file – интерфейс для выбора файла на клиентском компьютере для последующей отправки этого файла;

hidden – скрытое от пользователя поле формы, используемое для передачи дополнительных параметров веб-приложению. Эти параметры формируются на этапе создания html-документа, например, с помощью JavaScript;

image – серверное изображение на стороне клиента, т. е. поле с изображением. При нажатии на рисунок данные формы отправляются на сервер;

month – выбор месяца;

number – поле для ввода чисел;

password – строка для ввода пароля, обычное текстовое поле, но при вводе символы заменяются точками или звездочками;

radio – радиокнопка, разрешается выбрать только один вариант из набора;

range – ползунок для выбора чисел в указанном диапазоне;

reset – кнопка для очистки формы;

search – поле для поиска. Основное различие между текстовым полем и полем поиска заключается в том, как браузер его стилизует. Чаще всего поля поиска отображаются с закругленными углами и имеют крестик, который нужно нажать, чтобы очистить введенное значение;

submit – кнопка для отправки данных формы веб-приложению;

tel – поле для ввода телефонных номеров. Можно вводить и буквы и цифры. Отличие в семантическом наполнении;

text – строка редактирования (значение по умолчанию). Если в текстовое поле вводится текст с разрывами строки, браузер удаляет эти разрывы строк перед отправкой данных;

time – для времени;

url – для веб-адресов;

week – выбор недели.

Если атрибут type не задан, или задан, но ему присвоено значение не указанное в спецификации, то используется значение по умолчанию text.

Приведем пример формы с текстовыми полями (рисунок 12).

```
<body>
```

```
   form id="form1" name="form1" method="post" action="save.asp"
enctype="application/x-www-form-urlencoded">
     <p>
      <label for="fio">Ваше имя</label>
      <input   name="fio" id="fio" type="text" value="" size="50"
accesskey="f">
     </p>
     <p><label for="god">Год рождения</label>
        <input id="god" name="god" type="text" value="2005" size="4"
>
     </p>
     <p>
       <label for="prof">Род занятий</label>
       <input  id="prof" name="prof" type="text" value="студент">
     </p>
     <p><input type="submit" value="Отправить"></p>
   </form>
   </body>
 </html>
```

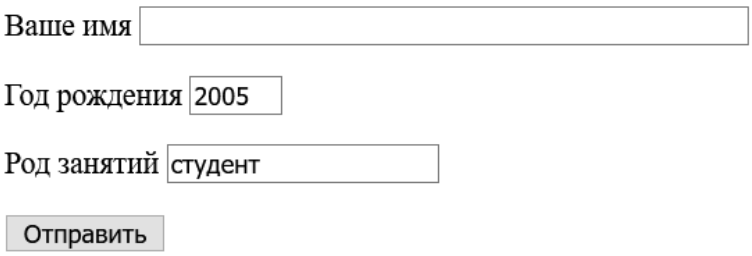


Рисунок 12 – форма

`<input type="text">` определяет поле ввода текста в одну строку. Ширина текстового поля по умолчанию составляет 20 символов. value – значение по умолчанию. Атрибут enctype определяется только для метода post. Значение по умолчанию. "application/x-www-form-urlencoded". Это значит, что все символы кодируются перед отправкой (пробелы преобразуются в символы «+», а специальные символы преобразуются в значения ASCII HEX).

По умолчанию при переходе по клавише Tab по полям формы перемещение происходит от первого элемента формы до последнего. Измените код предыдущего примера так, чтобы переходы по полям формы происходили в обратном порядке.

Приведем пример формы с радиокнопками (рисунок 13).

```
 <form>
 <h3>Самая высокая гора Африки</h3>
   <input type="radio" id="stenli" name="mountain" value="stenli">
   <label for="mountain">Стэнли</label><br>
   <input type="radio" id="kilim" name="mountain" value="kilim">
   <label for="kilim">Килиманджаро</label><br>
   <input type="radio" id="spic" name="mountain" value="spic">
   <label for="spic">Спик</label><br>
     <input type="radio" id="emin" name="mountain" value="emin">
   <label for="emin">Эмин</label><br>
     <input type="radio" id="lui" name="mountain" value="lui">
   <label for="lui">Луиджи ди Савойя</label>
```

```
</form>
```

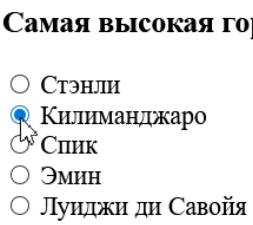


Рисунок 13 – форма

Приведем пример формы с переключателями (рисунок 14).

```
<form>
Что положить в классический салат "селедка под шубой"?<br>
  <input type="checkbox" id="part1" name="part1" value="Herring">
  <label for="part1">Сельдь</label><br>
  <input type="checkbox" id="part2" name="part2" value="Garlic">
    <label for="part2">Чеснок</label><br>
    <input type="checkbox" id="part3" name="part3" value="Beet">
  <label for="part3">Свекла</label><br>
  <input type="checkbox" id="part4" name="part4" value="Potato">
  <label for="part4">Картошка</label><br>
    <input type="checkbox" id="part5" name="part5" value="Onion">
  <label for="part5">Лук</label><br>
    <input type="checkbox" id="part6" name="part6" value="Boat">
  <label for="part6">Орехи</label><br>
    <input type="checkbox" id="part7" name="part7" value="Nuts">
  <label for="part7">Морковь</label><br>
    <input type="checkbox" id="part8" name="part8" value="Carrot">
  <label for="part8">Оливки</label><br>
    <input type="checkbox" id="part9" name="part9" value="Olives">
  <label for="part9">Куриное яйцо </label><br>
    <input type="checkbox" id="part10" name="part10" value="Egg">
  <label for="part10">Майонез</label>
</form>
```

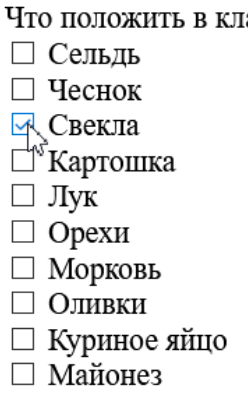


Рисунок 14 – форма

Приведем пример формы для ввода пароля.

Символы, вводимые в поле ввода пароля, заменяются звездочкой или жирной точкой. Пароль нельзя скопировать при помощи выделения и нажатия комбинации клавиш Ctrl+C. Значение пароля доступно для JavaScript и для web-приложения на сервере (рисунок 15).

```
<form name="form2" action="save.asp">
  <p>
```

```
  <b>Пароль</b> 
  <input   id="psw" name="psw" type="password">
  </p>
  <input type="submit" value="Отправить">
</form>
```

Извлечение пароля

```
<h3>Извлечение пароля</h3>
Пароль:
<input type="password" id="myPsw" value="123456">
<p>Нажми на кнопку, чтобы вывести пароль.</p>
<button onclick="getPsw()">Жми!</button>
<p id="demo"></p>
<script>
function getPsw() {
    var x = document.getElementById("myPsw").value;
    document.getElementById("demo").innerHTML = x;
}
</script>
```

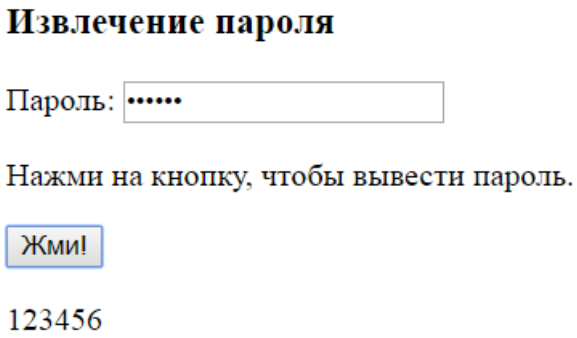


Рисунок 15 – форма с паролем

Атрибут checked

Флажок, принимающий значение true или false и сигнализирующий выбран ли элемент интерфейса. Используется только для флажков и радиокнопок. Значение по умолчанию – false.

Приведем пример формы с радиокнопками и переключателями. К форме из предыдущего раздела

```
<form id="form1" name="form1" method="post" action="save.asp">
  <p>
    <b>Ваше имя</b> 
    <input name="fio" type="text" value="" size="50"
    accesskey="f">
  </p>
  <p>
    <b>Год рождения</b> 
    <input id="god" name="god" type="text"
    value="1999" size="4" >
  </p>
  <p>
    <b>Род занятий</b> 
    <input    id="prof" name="prof" type="text"
    value="студент">
```

```
  </p>
  <p>
    <input type="submit" value="Отправить">
  </p>
</form>
```

добавим радиокнопки. Для этого атрибут type тега <input> установим в "radio". Радиокнопки позволяют пользователю выбрать только один из ограниченного числа вариантов. Радиогруппа должна иметь одно и то же имя (значение атрибута name), чтобы считаться группой.

```
<b>Пол</b><br>
<input   id="pol" name="pol" type="radio"
value="мужской"> <b>мужской</b><br>
<input id="pol" name="pol" type="radio"
value="женский"> <b>женский</b><br>
```

Затем добавим группу переключателей. Для этого атрибут type тега <input> установим в "checkbox". Группа переключателей позволяют пользователю выбрать сразу несколько вариантов.

```
<b>Какие языки вы знаете?</b><br>
<input type="checkbox" name="option1" value="a1" checked>русский<br>
<input type="checkbox" name="option2" value="a2"
checked>белорусский<br>
<input type="checkbox" name="option3" value="a3">английский<br>
<input type="checkbox" name="option4" value="a4">немецкий<br>
<input type="checkbox" name="option5" value="a5">французский<br>
```

Результат – на рисунке 16.

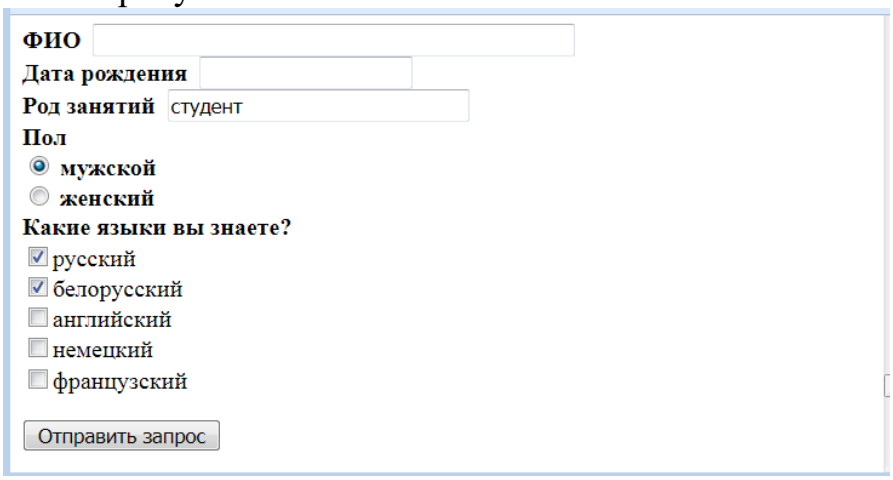


Рисунок 16 – форма

Программный выбор флажков

Используя javascript можно изменять значения атрибута checked программно. Для этого достаточно получить элемент, например, через id функцией getElementById.

```
var x = document.getElementById("product1");
```

А затем изменить его свойство checked.

```
x.checked = true;
```

Допустим у нас есть набор флажков для выбора конфет. Выберем только шоколадные конфеты. Выбор будем осуществлять через нажатие на кнопку.

```
<h3>Закажите конфеты</h3>
```

```
   <input type="checkbox" id="product1" value="gril"> Грильяж<br>
   <input type="checkbox" id="product2" value="iris"> Сливочный
ирис<br>
   <input type="checkbox" id="product3" value="red"> Красная
шапочка<br>
   <input type="checkbox" id="product4" value="min"> Минчанка<br>
    <button onclick="selectChocolate()">Выбрать только шоколадные
конфеты</button><br>
   <script>
 function selectChocolate() {
     var x = document.getElementById("product1");
     x.checked = true;
     x = document.getElementById("product3");
     x.checked = true;
     x = document.getElementById("product4");
     x.checked = true;
 }
 </script>
```

1.2.13. Атрибуты pattern, placeholder, required, title

В HTML5 были добавлены возможности для валидации полей формы. Рассмотрим валидацию на примере тега <input> с типом type="password".

```
 <form name="form2" action="autorize.asp">
   <p>
     <label for="psw">Пароль:</label>
     <input type="password" id="psw" placeholder="Пароль.."
pattern=".{6,}" title="Не менее шести символов" required>
   </p>
   <p>
     <input type="submit" value="Отправить">
   </p>
 </form>
```

Атрибут required тега <input> указывает, что поле ввода должно быть заполнено перед отправкой формы. Вид всплывающей подсказки и ее содержание могут разниться в различных браузерах. На рисунках 17-18 показано как выглядят подсказки в Firefox, Google Chrome.

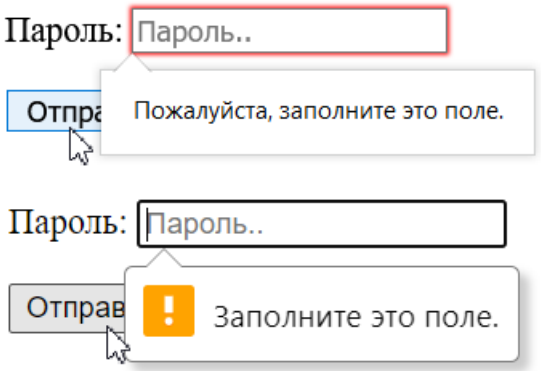


Рисунок 17 – пароли в Firefox

Глобальный атрибут title для тега <input> работает как всплывающая подсказка для формата ввода.

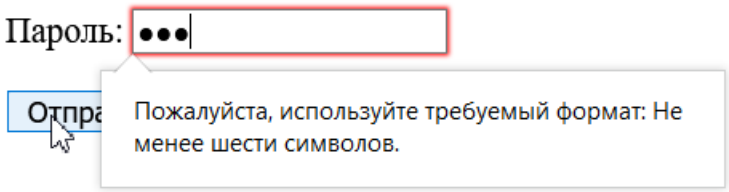


Рисунок 18 – пароли в Chrome

Атрибут placeholder выводит короткую подсказку, которая описывает ожидаемое значение элемента <input>.

Атрибут pattern задаёт регулярное выражение, при помощи которого будет проверяться введенное значение элемента <input>.

1.2.14. Скрытые поля

Можно определить на форме скрытое поле. Для этого в теге <input> тип поля устанавливается в hidden. Часто этот атрибут скрывает элемент управления, который не отображается браузером и не даёт пользователю изменять значения по умолчанию.

1.2.15. Кнопки

При конструировании формы создать кнопки можно различными способами.

Тип type="reset" для <input> задаёт кнопку для очистки формы. С точки зрения UX (user experience), это считается плохой практикой/

Тип type="submit" для <input> определяет кнопку для отправки данных формы на сервер. value задаёт надпись на кнопке. По умолчанию это «Submit» или «Отправить», в зависимости от языковых настроек.

Покажем, как получить доступ к кнопке submit.

```
 <h3>Демонстрация как получить доступ к кнопке Submit</h3>
 <input type="submit" id="mySubmit" value="Отправка данных формы">
 <p>Нажмите на кнопку "Извлечь текст", чтобы извлечь текст,
написанный на кнопке "Submit".</p>
 <button onclick="myFunction()">Извлечь текст</button>
 <p id="submitDemo"></p>
 <script>
  function myFunction() {
     var x = document.getElementById("mySubmit").value;
     document.getElementById("submitDemo").innerHTML = "Кнопка submit
подписана как \""+x+"\".";
 }
 </script>
```

Тип type="button" для <input> просто определяет кнопку без заданной функциональности. В атрибуте value определяет текст на кнопке.

Пример создания кнопки submit. При щелчке на кнопки вызывается скрипт createSubmit. После выполнения этого кода кнопка submit добавится в конец страницы.

```
<p>Нажмите на кнопку, чтобы создать элемент submit.</p>
<button onclick="createSubmit()">Создать</button>
<script>
 function createSubmit() {
    var x = document.createElement("INPUT");
    x.setAttribute("type", "submit");
    document.body.appendChild(x);
}
</script>
```

Тип type="image" для <input>– это серверное изображение на стороне клиента, т. е. поле с изображением (рисунок 19). При нажатии на рисунок данные формы отправляются на сервер, т.е. кнопка с изображением ведет себя как кнопка "submit". Но отправляются не значения полей, а координаты X и Y щелчка по изображению. Например, когда вы щелкаете изображение, вы отправляетесь по URL-адресу, подобному следующему: http://foo.com?pos.x=10&pos.y=35.

Благодаря этому удобно использовать изображения как кнопки для карты кликов.

```
<form action="demo_form.asp">
  <h1>Какая собака вам подходит по характеру?</h1>
  <input type="image" src="images/dogs.jpg" alt="Submit">
</form>
```

Рисунок 19 – input type="image"

Многострочное текстовое поле <textarea>

Если в форме требуется ввести не одну, а несколько строк информации, например, при сборе пользовательских данных, таких как комментарии или обзоры, то используется тег <textarea>.

Основное различие между текстовым полем textarea и обычным однострочным текстовым полем input type="text" заключается в том, что пользователям разрешено вводить текст, который включает символы перехода на следующую строку. По умолчанию тег создаёт пустое поле шириной в 20 символов, состоящее из двух строк. Выводимый текст находится между тегами. Тре-

буется закрывающий тег. Если нужно определить значение по умолчанию для <textarea>, то можно поместить это начальное значение между открывающим и закрывающим тегами:

```
<textarea>
  по умолчанию в этом элементе находится этот текст
</textarea>
```

Атрибуты тега <textarea>

Атрибут rows задаёт количество видимых строк. Если содержимое элемента выходит за пределы зоны видимости возникает полоса прокрутки.

Атрибут cols задаёт количество столбцов символов в символах средней ширины. Вводимый текст автоматически переносится на следующую строку для того, чтобы были видимыми длинные строки без необходимости прокрутки. Если содержимое элемента выходит за пределы зоны видимости возникает полоса прокрутки.

Атрибут-флаг readonly указывает что, текстовое поле не может изменяться пользователем, в том числе вводиться новый текст или модифицироваться существующий. Кроме того, такое поле не может получить фокус путем нажатия на клавишу Tab, мышью или другим способом. Тем не менее, состояние и содержимое поля можно менять с помощью скриптов.

Атрибут wrap устанавливает правила переноса текста в поле <textarea> и вид, в каком данные будут отправлены на сервер. Если этот параметр отсутствует, текст в поле набирается одной строкой, когда число введенных символов превышает ширину области, появляется горизонтальная полоса прокрутки. Нажатие клавиши ввода переносит текст на новую строку, и курсор устанавливается у левого края поля. Может принимать следующие значения:

soft – длинный текст, который самостоятельно не помещается в поле по ширине, будет автоматически перенесен на новую строку, однако передаваться на сервер будет как одна строка. Нажатие клавиши ввода на клавиатуре устанавливает перенос текста, который сохраняется при отправке формы.

hard – слова в поле переносятся механически, чтобы они поместились в размер области, и при отправке на сервер точки автоматического переноса сохраняются.

Атрибут required указывает, что текстовая область обязательно должна быть заполнена.

Атрибут placeholder определяет текст-заполнитель, представляющий собой краткую подсказку, которая описывает ожидаемое значение текстовой области.

Атрибут maxlength определяет максимальное количество символов, разрешенное в текстовой области.

Атрибут dirname указывает, что направление текста текстового поля будет отправлено.

Приведем пример использования текстовой области (рисунок 20).

```
<form id="opros" name="opros" action="save.asp" method="get">
  <p>Наш программный продукт "Планировщик задач"</p>
  <p>
    <textarea id="comment" name="comment" rows="3"
```

```
    cols="55" disabled="disabled">
     Вы стали активным пользователем нашего программного
     продукта 20/06/19. За это время вам была
     предоставлена новая версия программы.
    </textarea>
  </p>
  <p>Ваши комментарии по поводу использования нашей программы:</p>
  <p>
    <textarea id="comment" name="comment" rows="4"
    cols="55"></textarea>
  </p>
  <input id="btnSave" name="btnSave" type="submit"
  value="Сохранить">
</form>
```

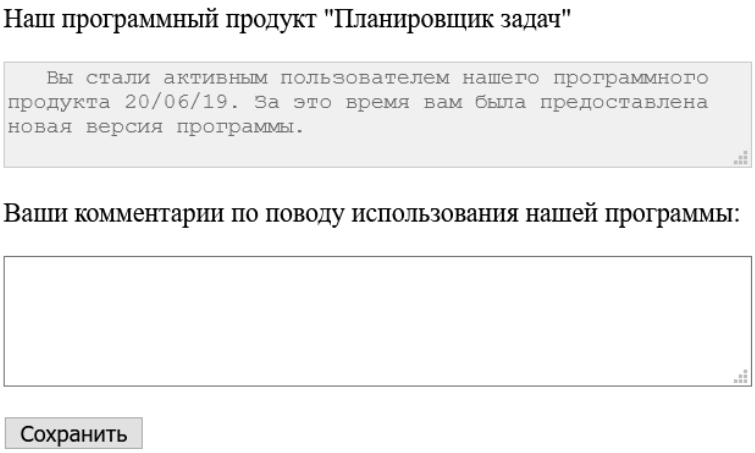


Рисунок 20 – textarea

1.2.16. Списки. <select>, <datalist>, <optgroup> и <option>

С помощью тега <select> можно создать элемент интерфейса в виде выпадающего списка, а также список с одним или множественным выбором. Конечный вид списка зависит от использования параметра size тега <select>, который устанавливает высоту списка. Ширина списка определяется самым широким текстом, указанным в теге <option>, а также может изменяться с помощью стилей. Каждый пункт создаётся с помощью тега <option>, который должен быть вложен в контейнер <select>. Если планируется отправлять данные списка на сервер, то требуется поместить элемент <select> внутрь формы. Это также необходимо, когда к данным списка идет обращение через скрипты. Тег <option> используется для задания отдельных пунктов списка, создаваемого с помощью контейнеров <select>, <optgroup> или <datalist>.

Тег <datalist> также как и <select> определяет список, но используется внутри тега <input>. Пользователи будут видеть раскрывающийся список предопределенных параметров при вводе данных (рисунок 21).

При этом атрибут list тега <input> должен содержать id тега <datalist>.

```
<form action="/action_page.php">
  <input list="browsers" name="browser">
  <datalist id="browsers">
    <option value="Internet Explorer">
```

```
    <option value="Firefox">
    <option value="Chrome">
    <option value="Opera">
    <option value="Safari">
  </datalist>
  <input type="submit" value="Выбрать браузер">
</form>
```

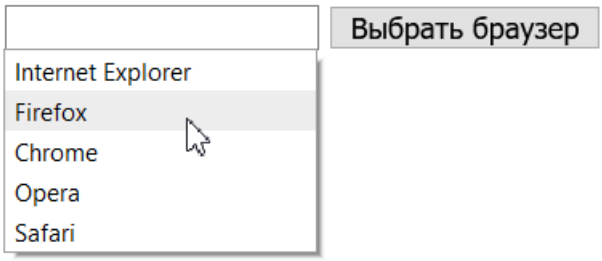


Рисунок 21 – select

Атрибуты тега <select>

Кроме атрибутов присущим всем элементам формы у тега <select> есть дополнительно атрибуты size, multiple и required.

Атрибут size устанавливает число видимых элементов списка и тем самым вид списка: если `size="1"`, то создаётся раскрывающийся список, во всех остальных случаях создаётся развернутый список.

Атрибут-флаг multiple определяет, разрешен ли множественный выбор элементов списка.

Атрибуты тега <option>

Атрибут-флаг selected, определяет, что данный пункт списка выделен по умолчанию.

Атрибут label задаёт метку, которая используется для формирования иерархии списка.

Value определяет значение, которое будет отправлено на сервер, в случае выбора текущего пункта списка. На сервер отправляется пара «имя=значение», где имя задаётся параметром name тега <select>, а значение – параметром value выделенных пунктов списка. Значение может, как совпадать с текстом пункта, так быть и самостоятельным. Параметр value применяется также для получения значений данных через скрипты.

Disabled определяет будет ли элемент доступен для пользователя.

Приведем пример создания списков.

```
<form id="form1" name="form1" action="save.asp"
 method="get" enctype="application/x-www-form-urlencoded" >
  "Ромашка" предлагает с доставкой:<br>
   <select id="menu" name="menu" size="4" multiple="multiple">
    <option value="1" selected="selected">Пицца</option>
    <option value="2">Пончики</option>
    <option value="3" selected="selected">Круассаны</option>
    <option value="4">Шаурма</option>
    <option value="5">Бутерброды</option>
  </select><br>
```

```
  Напитки<br>
   <select id="toys" name="toys" size="1">
    <option value="1">Минеральная вода</option>
    <option value="2" selected="selected">Кофе</option>
    <option value="3">Кола</option>
  </select>   
  <input type="submit" value="Заказать" />
</form>
```

Тег <optgroup> предназначен для создания многоуровневого списка и представляет собой контейнер, внутри которого располагаются теги <option> объединенные в одну группу. Особенностью тега <optgroup> является то, что он выделяется с помощью полужирного начертания, а все элементы, входящие в этот контейнер, смещаются вправо от своего исходного положения. Иерархия в списке создаётся при помощи атрибутов label тегов <optgroup> и <option>. В атрибуте label тега <optgroup> записывается текст, который будет входить в список в виде заголовка группы, а в атрибуте label тега <option> – элемента группы.

Приведем пример двухуровневого списка (рисунок 22)

```
<form action="/action_page.php">
  <label for="menu">Вы можете получить один товар со скидкой
50%</label>
  <select name="menu" id="menu">
    <optgroup label="Обувь">
      <option value="b11">Босоножки</option>
      <option value="b12">Кеды</option>
      <option value="b13">Балетки</option>
    </optgroup>
    <optgroup label="Майки">
      <option value="b21">Спортивные</option>
      <option value="b22">Майки с капюшоном</option>
    </optgroup>
  </select>
  <br><br>
  <input type="submit" value="Заказать">
</form>
```

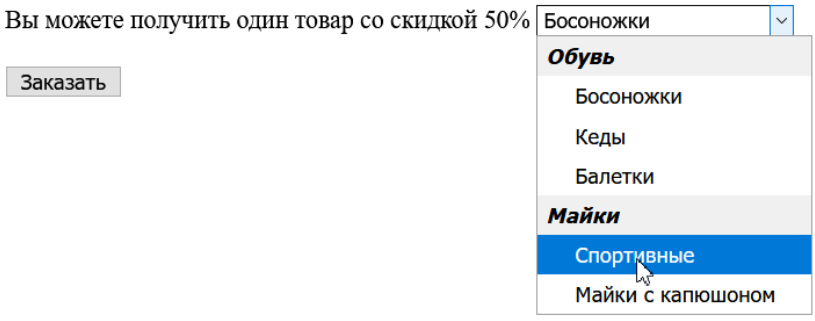


Рисунок 22 – двухуровневый список

Группы. <fieldset>, <legend>

Тег <fieldset> предназначен для стилистической и семантической группировки элементов формы. Такая группировка облегчает работу с формами, со-

держащих большое число данных. Например, один блок может быть предназначен для идентификации пользователя, а другой – для группы радиокнопок. Для наглядности браузеры отображают результат использования тега <fieldset> в виде рамки. Тег <fieldset> не имеет атрибутов.

Тег <legend> формирует надпись на рамке вокруг группы. Он должен идти первым в теге <fieldset>.

Часть программ чтения с экрана произносят заголовок формы <legend> перед произношением названия меток элементов.

```
 <form>
   <fieldset>
     <legend>Представьтесь</legend>
     <p>
       <label for="name3">Ваша фамилия</label>
       <input name="name3" id="name3">
     </p>
     <p>
       <label for="name1">Имя</label>
       <input name="name1" id="name1">
     </p>
     <p>
       <label for="name2">Отчество</label>
       <input name="name2" id="name2">
     </p>
   </fieldset>
   <fieldset>
     <legend>Выберите форму участия</legend>
     <p>
       <input type="radio" name="size" id="size_1" value="room">
       <label for="room">Выступление на заседании в зале</label>
     </p>
     <p>
       <input type="radio" name="size" id="size_2" value="dist">
       <label for="dist">Выступление с использованием дистанционных
технологий</label>
     </p>
     <p>
       <input type="radio" name="size" id="size_3" value="print">
       <label for="print">Предоставление доклада в письменном виде
для печати</label>
     </p>
   </fieldset>
 </form>
```

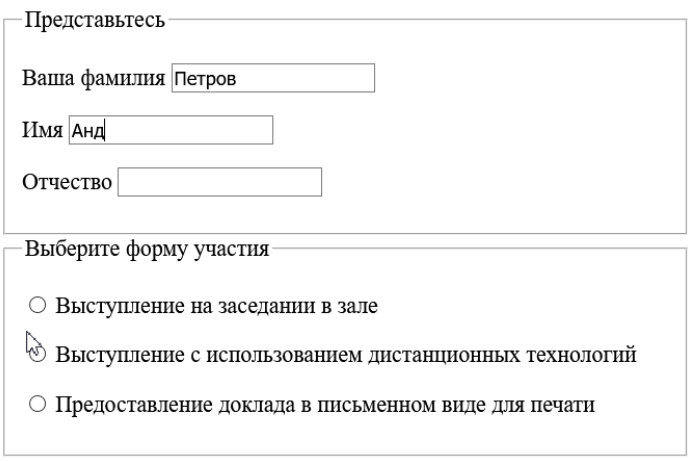


Рисунок 24 – список с fieldset

Структурирование форм

Общепринятой практикой является оборачивать элемент формы и соответствующую ему метку в блочный тег или <p>. Можно также использовать для структурирования элементов формы списки, так чаще поступают с флажками или радио-кнопками.

Если форма достаточно большая, то каждый отдельный раздел функциональности можно помещать в теги <section> с элементами <fieldset>.

Ниже (рисунок 25) приведен пример формы оплаты, с разделение на секции. Внутри тега <form> есть заголовок первого уровня и параграф, информирующий пользователей о том, как помечены поля, обязательные для заполнения. Поля с контактной информацией обернуты в отдельный элемент <section>. Заголовок второго уровня содержит информацию о содержании секции. В раздел входит три текстовых поля: для ввода ФИО, e-mail и пароля. С каждым текстовым полем связан соответствующий тег <label>. Каждое текстовое поле и связанная с ним метка обернуты в тег параграфа. Все поля, связанные с реквизитами оплаты, выделены еще в одну секцию. Сумма к оплате и кнопка также выделены в отдельные секции. Таким образом вся форма поделена на четыре семантических части.

```
<!doctype html>
 <html>
  <head>
   <meta charset="utf-8">
   <title>форма</title>
   <style>
    h1   { margin-top: 0;       }
    form {
          margin: 0 auto;
          width: 400px;
          padding: 1em;
          border: 1px solid #CCC;
          border-radius: 1em;
         }
label span {
            display: inline-block;
```

```
            width: 120px;
            text-align: right;
           }
 input {
        font: 1em sans-serif;
        width: 250px;
        box-sizing: border-box;
        border: 1px solid #999;
       }
 input:focus {border-color: #000;            }
 button {margin: 20px 0 0 124px;       }
 label  {position: relative;      }
 label em {
          position: absolute;
          right: 5px;
          top: 20px;
         }
    </style>
   </head>
  <body>
  <form method="post">
   <h1>Форма оплаты</h1>
   <p>Поля, обязательные для заполнения, помечены <strong><abbr
title="Обязательно к заполнению">*</abbr></strong>.</p>
   <section>
    <h2>Контактная информация</h2>
    <p>
     <label for="fio">
      <span>ФИО: </span>
      <strong>
       <abbr title="Обязательно к заполнению">*</abbr>
      </strong>
     </label>
     <input type="text" id="fio" name="fio">
    </p>
    <p>
     <label for="mail">
      <span>E-mail: </span>
      <strong>
       <abbr title="Обязательно к заполнению">*</abbr>
      </strong>
     </label>
     <input type="email" id="mail" name="usermail">
    </p>
    <p>
     <label for="pwd">
      <span>Пароль: </span>
      <strong>
       <abbr title="Обязательно к заполнению">*</abbr>
      </strong>
     </label>
     <input type="password" id="pwd" name="password">
    </p>
```

```
  </section>
  <section>
   <h2>Платежные реквизиты</h2>
    <p>
     <label for="card"><span>Тип карты:</span></label>
     <select id="card" name="usercard">
      <option value="visa">Visa</option>
      <option value="mc">Mastercard</option>
      <option value="belcart">БЕЛКАРТ</option>
     </select>
   </p>
   <p>
    <label for="number">
     <span>Номер карты:</span>
     <strong>
      <abbr title="Обязательно к заполнению">*</abbr>
     </strong>
    </label>
    <input type="tel" id="number" name="cardnumber">
   </p>
   <p>
    <label for="date">
     <span>Действует до:</span>
     <strong>
      <abbr title="Обязательно к заполнению">*</abbr>
     </strong>
    </label>
    <input type="date" id="date" name="date">
   </p>
  </section>
   <section>
   <h2>Сумма к оплате</h2>
   <p>
    <label for="sum">
     <span>К оплате: </span>
     <strong>
      <abbr title="Обязательно к заполнению">*</abbr>
     </strong>
     <input type="number" id="sum" name="sum" value="0"
     min="0" max="10000" step="0.01">
    </label>
   </p>
   </section>
   <section>
    <p><button type="submit">Оплатить</button> </p>
   </section>
  </form>
 </body>
</html>
```

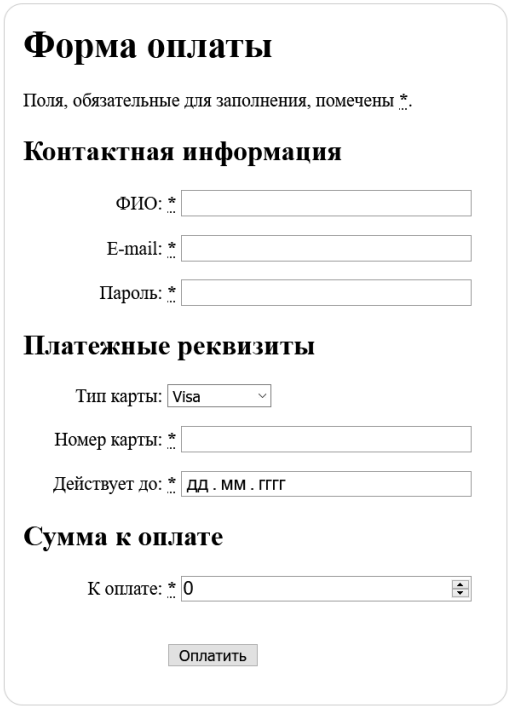


Рисунок 25 – форма оплаты

1.3. Каскадные таблицы стилей CSS

1.3.1. CSS. Спецификация, версии языка

Каскадные таблицы стилей или ***CSS (****Cascading Style Sheets****)*** – это язык описания таблиц стилей, позволяющий применять стили в структурированных документах, например, в HTML- и XML-документах, а также задавать свойства элементов разметки (тегов), которые не могут быть заданы с помощью стандартных атрибутов html-тегов. CSS позволяет приводить к единому внешнему виду целые группы html-документов, даже если это и не было предусмотрено при их создании. Это делает стиль представления документов независимым от их содержания, что существенно упрощает разработку веб-страниц и поддержку сайтов.

На сайте w3schools.com есть прекрасная демонстрация применения различных стилей к простейшей странице https://www.w3schools.com/css/css_intro.asp.

В 1994 году Хокон Виум Ли из Норвегии предложил концепцию каскадных таблиц стилей. CSS был разработан и дальше разрабатывается рабочей группой W3C, которая называется CSS Working Group. Эта группа состоит из представителей производителей браузеров и других компаний, которые заинтересованы в развитии CSS и приглашенных экспертов.

Новые возможности CSS разрабатываются и возникают по различным причинам: это может быть запрос какого-либо конкретного браузера, который хочет иметь определенные возможности; это может быть запрос веб-дизайнеров и разработчиков; обобщение уже реализованных возможностей в браузерах; решение самой рабочей группы. CSS постоянно развивается, появляются новые функции. Основным требованием к обновлениям является то, что

все старые сайты, созданные с использованием старых версий CSS, доступных на время создания сайта, должны работать без изменения дизайна.

CSS стандартизован спецификацией W3C https://www.w3.org/Style/CSS/#specs. Стандарт CSS делится на уровни: CSS1, CSS2.1 и CSS3.

Спецификация CSS все время расширяется и, как следствие, увеличивается в объеме. Описание версии CSS 2.1 было в пять раз больше по сравнению с описанием предыдущей версии, поэтому в стандарте CSS3 все функции распределили по наборам индивидуальных стандартов, называющихся модулями. Всем этим новым модулям CSS было присвоено коллективное название CSS3. В настоящее время действует стандарт CSS3.

Спецификация CSS3 содержит около 50 модулей разной степени завершенности. Эти модули содержат как возможности, поддерживаемые самыми последними версиями всех современных браузеров, так и экспериментальные возможности.

Очень редко бывает так, что новое свойство, определенное в CSS, оказывается сразу реализованным во всех браузерах, поэтому для каждого нового свойства нужно проверять, какие браузеры его поддерживают.

1.3.2. Полезные ссылки

CSS спецификация стандарт W3C:
https://www.w3.org/Style/CSS/#specs.
Справочник по поддержке браузерами свойств CSS:
https://www.w3schools.com/cssref/css3_browsersupport.asp.
Учебники:
http://www.w3schools.com/css/default.asp/
http://http://htmlbook.ru/samcss
https://webref.ru/course/css-basics
https://developer.mozilla.org/ru/docs/Learn/CSS.
https://professorweb.ru/my/css/css_theory/level1/css_index.php.
https://metanit.com/web/html5/5.1.php.
Справочники:
http://http://www.w3schools.com/cssref/default.asp
http://http://htmlbook.ru/css
https://developer.mozilla.org/ru/docs/Web/CSS/Reference
Практикумы:
https://htmlacademy.ru/
https://www.codecademy.com/learn

1.3.3. Типы стилевых таблиц

Основная особенность CSS заключается в том, что таблицы стилей объединяются, и именно результирующее правило в итоге определяет окончательное представление документа.

Когда браузер загружает html-документ он читает таблицу стилей и форматирует страницу в соответствии с информацией в таблице стилей. Существуют три способа добавления таблицы стилей к документу:

внешние CSS таблицы;

внутренние CSS таблицы;

встроенный CSS стили.

Как видно таблицы стилей делятся на группы по расположению.

Первая группа – ***внешние*** таблицы стилей, их еще иногда называют связанные. Внешние таблицы стилей расположены в отдельном файле с расширением ****.css***. Такие таблицы стилей могут применяться к любому html-документу, к которому они подключены.

Вторая группа – ***внедренные*** таблицы стилей. Внедренные таблицы стилей расположены внутри html-документа, в его головной части, в теге <style>. Правила, описанные в такой таблице, распространяются на элементы разметки всего документа. Третья группа – ***встроенные*** таблицы стилей. Встроенные таблицы стилей вставляются непосредственно в элемент разметки, посредством атрибута style. Область применения таких стилей только тот тег, в котором они определены.

1.3.4. Внешние таблицы стилей

Внешние таблицы стилей являются наиболее распространенным и одновременно имеющим наибольшую общность типом таблиц стилей. Связанная таблица стилей располагается в отдельном файле, имеющем расширение *.css. Такие стилевые таблицы могут быть одновременно подключены к целому ряду html-документов. Изменение стилевого правила в такой таблице повлечет изменение дизайна всех веб-страниц, к которому подключена внешняя таблица стилей. К одному html-документу могут быть подключены несколько внешних стилевых таблиц. В одном html-документе могут одновременно работать стилевые правила, определенные и во внешней, и во внутренней, и во встроенных таблицах стилей. Внешние таблицы стилей могут создаваться в любом текстовом редакторе. Они подключаются к html-документу в теге <link>**.** Для этого в теге <link> значение атрибута rel должно быть установлено в stylesheets, а в атрибуте href должен быть указан сетевой адрес таблицы стилей. Внешний файл .css не должен содержать никаких html-тегов. Ниже приведен пример подключения внешнего стилевого файла style1.css, который расположен на диске в той же директории, что и веб-страница. Первой строкой в стилевом файле записано @-правило (at-rules), @-правило всегда добавляется первым в файл, каждое из правил имеет свой синтаксис. Правило @charset определяет кодировку символов, используемую в таблице стилей.

Внешние таблицы стилей позволяют веб-страницам загружаться быстрее. Когда используются такие таблицы стилей, веб-страницы содержат только html-код, без громоздкого кода внутренних таблиц стилей и без встроенных стилей. Когда браузер загрузит внешнюю таблицу стилей при загрузке первой

страницы сайта, он сохранит этот файл на клиентском компьютере посетителя веб-страницы в специальной системной папке, называемой кэшем, для быстрого доступа к нему. Когда посетитель веб-страницы переходит к другим страницам сайта, которые используют ту же внешнюю таблицу стилей, браузеру уже нет необходимости снова загружать таблицу стилей. Он попросту загружает запрашиваемый html-файл и использует внешнюю таблицу стилей из своего кэша, что даёт существенный выигрыш во времени загрузки страниц.

1.3.5. Внутренние таблицы стилей

Внутренние таблицы стилей, иногда их еще называют внедренными, располагаются прямо в html-документе, а правила, описанные в них, распространяются на весь документ. В такой таблице размещают уникальные стили, предназначенные только для этой страницы.

Внутренняя таблица стилей помещается внутри тега <style>. Тег <style> помещается в заголовочную часть html-документа.

Применение внутренних таблиц стилей не так эффективно, как использование внешних таблиц: если потребуется внести изменения, то будет необходимо прописывать стилевые правила отдельно для каждой страницы.

Можно поместить элемент style и все его стили после элемента title, но веб-дизайнеры обычно размещают их прямо перед закрывающим тегом </head>. Однако, если на странице используется javascript-код, он должен располагаться после таблиц стилей. Часто javascript-сценарии используют CSS, поэтому, добавляя таблицы стилей первыми, можно гарантировать, что javascript-код будет иметь все необходимые для своего выполнения данные.

1.3.6. Встроенные стили

Встроенные стили имеют наименьшую область применения. Их область действия распространяется только на тот тег, в котором они определены. Для определения стиля используется атрибут style. Встроенные стили активно используются в генераторах html-кода. Если на странице много встроенных таблиц стилей, то применение связанных таблиц стилей к такой странице может не принести должного эффекта. Встроенные стили категорически не рекомендуется использовать. Их применение делает код трудным для технического обслуживания при внесении изменений и кроме того тяжелым для чтения и понимания. Применение встроенных стилей оправдано только в том случае, если такое правило нужно применить только к одному тегу одного документа и нужно перекрыть CSS правила из других способов подключения или в тестовых целях. Причем для элемента определено много противоречивых правил во внешних и внутренних стилевых таблицах, и достаточно долго просчитывать приоритет стилей по правилам каскадности. В данном случае для всего документа определен темно-синий цвет фона и белый цвет шрифта текста в элементе <body>, Во вложенном теге эти свойства переопределены на желтый и голубой соответственно. Поскольку при сложении стилей действует принцип «своя рубашка

ближе к телу» очевидно, что для блока действуют правила, определенные непосредственно в нем (рисунок 26).

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="style.css">
  </head>
  <body style="background-color:darkblue; color:white">
    Мне нравится белый текст на темном фоне.
    <div style="background-color:yellow; color:blue;
    border-style:dashed;  border:4; border-color:crimson">
    Но иногда удобнее читать на светлом фоне
  </div>
</body>
</html>
```

Рисунок 26 – оформление текста стилями

Каскадность

Каскадность – это набор правил, который определяет, каким образом браузер должен отображать многократно определенные стили, относящиеся к одному и тому тегу.

Если некоторые свойства были определены для одного и того же селектора (элемента) в разных таблицах стилей, будет использоваться значение из последней прочитанной таблицы стилей.

Например, если есть внешняя таблица стилей style.css, в которой определен стиль для параграфа темно-синий фон:

```
h1 {
    background-color: darkblue;
}
```

А во внутренней таблице стилей для параграфа задан оранжевый фон.

```
<style>
h1 {    color: orange;}
</style>
```

Если в html-документе сначала подключен внешний файл, а потом определена внутренняя таблица стилей, то фон будет как во внутренней таблице – оранжевый.

```
<link rel="stylesheet" href="style.css">
<style>
  h1 {      color: orange;}
</style>
```

Если в html-документе сначала определена внутренняя таблица стилей, а потом подключен внешний файл, то фон будет как во внешней таблице – темно-синий.

Если для html-элемента задано более одного стиля, то при обработке стилей браузер следует двум основным правилам:

сначала все правила стилей, определенные для элементов страницы складываются, (они могут быть описаны и в различных таблицах) и только после этого применяется итоговый стиль;

при сложении стилей приоритет имеют атрибуты, определенные в стиле с более высоким приоритетом.

По способу подключения по приоритетности стили можно разделить на три группы:

1. Высший приоритет имеют встроенные стили, определенные в теге через атрибут style.
2. Потом идут стили из внешних и внутренних таблиц стилей с учетом порядка их подключения в разделе заголовка.
3. Самый низкий приоритет имеют стили браузера.

Таким образом, встроенный стиль имеет наивысший приоритет и переопределяет внешние и внутренние стили и настройки браузера по умолчанию.

1.3.7. Модификатор !important

Вес правила можно задать с помощью ключевого слова !important, которое добавляется сразу после значения свойства, например:

```
p {   font-weight: bold!important;}
```

Правило необходимо размещать в конце объявления перед закрывающей скобкой, без пробела. Такое объявление будет иметь приоритет над всеми остальными правилами. Модификатор !important усложняет отладку и нарушает естественное каскадирование стилей, поэтому его использование не рекомендуется. Однако этот способ позволяет отменить значение свойства и установить новое для элемента из группы элементов в случае, когда нет прямого доступа к файлу со стилями.

1.3.8. Специфичность

В рамках одного способа подключения приоритет CSS-стилей определяется расчетом специфичности https://www.w3.org/TR/selectors/#specificity. Специфичность – это способ, с помощью которого браузеры определяют, какие значения свойств CSS наиболее соответствуют элементу и, следовательно, будут применены. Специфичность выражается числом, который рассчитывается для каждого селектора. Это число представляет собой вес, придаваемый конкретному стилевому правилу. Чем показатель специфичности больше, тем приоритет выше. В теории чем выше специфичность, тем более ограниченное количество тегов выбирает селектор.

При расчете специфичности третий разряд числа представляет собой количество id в селекторе (#example).

Второй – количество селекторов классов (например, .example), селекторы атрибутов (например, [type="email"]) и псевдоклассов (например, :hover).

Первый разряд числа – это количество html-элементов (p, div, a…) или псевдоэлементов (::after).

Разряд			
Коэффициент			

Универсальный селектор (*), комбинаторы (+, >, ~, ' ') и отрицающий псевдокласс (:not()) не влияют на специфичность.

Приведем примеры расчета специфичности:

Спецификатор	Коэффициент специфичности
aside h1	0-0-2
*#price	1-0-0
#price	1-0-0
ol li a.book	0-1-3

```
<html>
<head>
<title> Специфичность </title>
<style >
div #main p a {color: orange;}
#main a {color: green;}
.container #main a {color: pink;}
p a {color: yellow; }
a {color: black; }
</style>
<body>
<div class="container">
  <div id="main">
    <p>
      <a href="#">Hello! Розового цвета!!!</a>
    </p>
  </div>
 </div>
<p> как рассчитывается специфичность в этом примере:
(для удобства ставим запятые)<pre>
div #main p a: 1,0,3
#main a:       1,0,1
.container #main a: 1,1,1
p a:           0,0,2
a:             0,0,1
</pre></p>
</body></html>
```

Если при объявлении стиля используется модификатор !important, то тогда этот стиль получает наивысший приоритет среди всех прочих объявлений. Хотя технически модификатор !important не имеет со специфичностью ничего общего, он непосредственно на нее влияет.

Отрицающий псевдокласс :not не учитывается как псевдокласс при расчете специфичности. Однако селекторы, расположенные внутри :not, при подсчете количества по типам селекторов рассматриваются как обычные селекторы и учитываются. Так, например, оба стиля, приведенных ниже, имеют одинаковую специфичность 012.

```
div.outer p {color: orange;}
div:not(.outer) p {color: lime;}
```

Для подсчета специфичности можно пользоваться онлайн калькуляторами специфичности, например, https://specificity.keegan.st/ (рисунок 27).

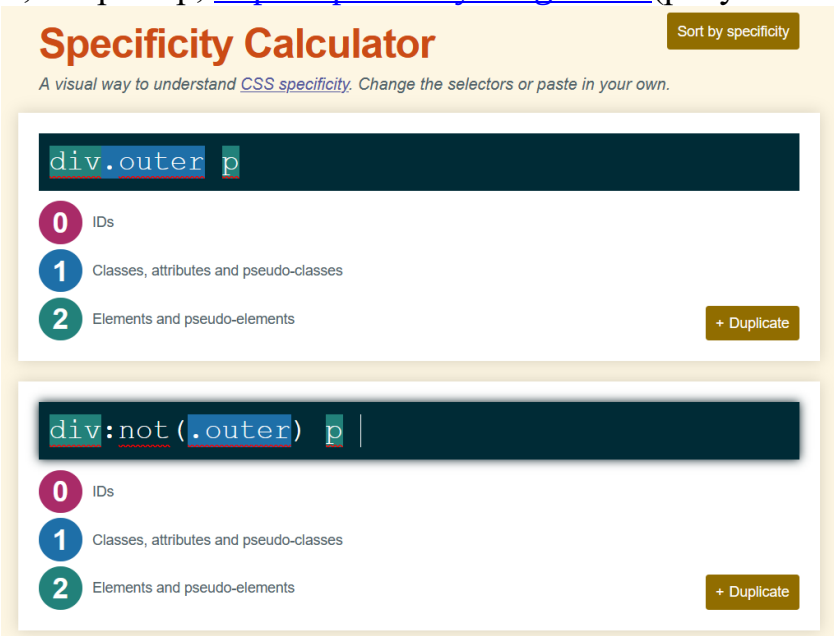


Рисунок 27 – Онлайн калькулятор специфичности.

При наличии разных правил с одинаковой специфичностью действует объявленное последним.

```
div p {color: red; border: 1px solid black;}
body p {text-align: center; color: yellow;}
p + p {background-color: blue; color: pink;}
```

Для всех трех селекторов коэффициент специфичности 0-0-2 (рисунок 28).

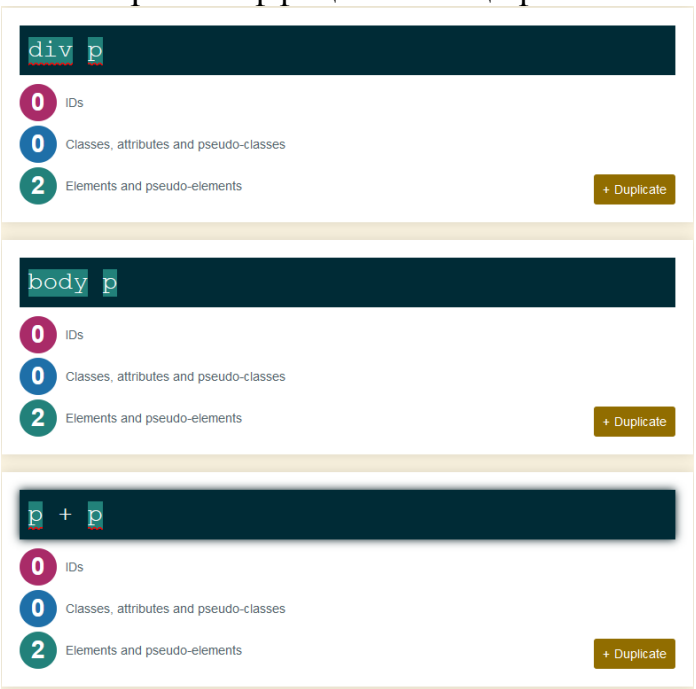


Рисунок 28 –калькулятор специфичности.

Для тега <p>, который захватится всеми этими селекторами сформируется следующий список свойств:

```
border: 1px solid black;
text-align: center;
background-color: blue;
```

```
color: pink;
```

1.3.9. Наследование

Наследование – это механизмом, с помощью которого определенные свойства передаются от предка к его потомкам.

Спецификацией CSS предусмотрено наследование свойств, относящихся к текстовому содержимому страницы, таких как color, font, letter-spacing, line-height, list-style, text-align, text-indent, text-transform, visibility, white-space и word-spacing. Во многих случаях это удобно, так как не нужно многократно задавать повторяющиеся стили для вложенных элементов, например, размер шрифта и семейство шрифтов для каждого элемента веб-страницы.

Но наследуются не все свойства. Например, свойства, относящиеся к форматированию блоков, не наследуются. Это background, border, display, float и clear, height и width, margin, min-max-height и -width, outline, overflow, padding, position, text-decoration, vertical-align и z-index.

Можно принудить элемент наследовать любое значение свойства родительского элемента. Для этого используется ключевое слово inherit. Принудительное наследование свойства родителя при помощи ключевого слова inherit работает даже для тех свойств, которые не наследуются по умолчанию.

1.3.10. Просмотр стиля в Инструментах разработчика браузера

К одному элементу могут применяться стили из разных источников. Проверить, какие стили применяются, можно в режиме разработчика браузера. Для этого над элементом нужно щелкнуть правой кнопкой мыши и выбрать пункт «Исследовать элемент», «Посмотреть код элемента» или что-то аналогичное в зависимости от браузера. Откроются Инструменты разработчика. В правом столбце будут перечислены все свойства, которые заданы для этого элемента или наследуются от родительского элемента.

1.3.11. Синтаксические правила CSS

CSS – это язык на основе правил: С помощью CSS задаются правила, определяющие группы стилей, которые должны применяться к определенным элементам или группам элементов на веб-странице. Файлы, содержащие каскадные таблицы стилей – это текстовые файлы, имеющие расширение *.css. Каждая таблица стилей состоит из одного или нескольких правил. Правило в CSS состоит из двух частей: селектора и описания. ***Селектор*** – это название элемента разметки, к которому будет применяться правило, а ***описание*** – это задание значений одному или более свойствам выбранного элемента разметки. Описание заключается в фигурные скобки. Каждое определение, входящее в описание, в свою очередь также состоит из двух частей: ***свойства*** и ***значения***. Свойство отделяется от значения двоеточием. CSS-свойства имеют разные допустимые значения в зависимости от того, какое свойство указывается. Если описание содержит определение более чем одного свойства, то определения

различных свойств отделяются друг от друга точкой с запятой. После последнего определения точку с запятой ставить не обязательно, но это не является ошибкой.

Формат правила имеет следующий вид:

```
селектор { свойство : значение}
```

или, для нескольких свойств:

```
селектор
        {
          свойство1 : значение1;
          свойство2 : значение2;
                    …
          свойствоN : значениеN
         }
```

Например, для тега параграфа <p> устанавливается желтый цвет фона, синий цвет текста и размер шрифта 18 пикселей.

```
p{
  background:yellow;
  font-size:18px;
  color:blue}
```

Таблицы стилей, так же как и HTML, не чувствительны к регистру, т. е. строчные и прописные символы не различаются, за исключением тех частей стилевых правил, которые не являются объектами языка CSS.

Комментарии начинаются с символов *"/*"* и заканчиваются символами *"*/"*.

Значения свойств, являющихся адресами, устанавливаются следующим образом:

```
url(адрес)
```

Например, при установке изображения bg.gif в качестве фона для блока тега правило имеет вид:

```
div { background:url(bg.gif) }
```

При установке различных свойств одному объекту свойства могут группироваться, при этом значения свойств в группе оделяются друг от друга пробелом. Например, следующие правила для установки свойств шрифта в теле html-документа эквивалентны:

```
body
 {
  font-family:Arial, Helvetica, sans-serif;
  font-size:10px;
  font-style:italic; }
```

и

```
body { font:Arial, Helvetica, sans-serif 10px italic }
```

Первое правило использовать предпочтительнее, т. к. если вы сделали ошибку в назначении свойства, то строка, в которой есть незнакомое для браузера значение свойства, проигнорируется. Таким образом, в первом случае будет проигнорировано одно свойство из трех, а во втором случае – все три свойства.

1.3.12. Селекторы

Селектор – это формальное описание элемента HTML или группы элементов, или части элемента. Селектор определяет, к каким элементам DOM будет применяться стилевое правило.

Селекторы по сущностям, участвующим в описании, условно можно разбить на группы:

- универсальный селектор;
- селекторы элементов;
- селекторы классов и id;
- селекторы на основе их расположения;
- селекторы атрибутов;
- селекторы псевдоклассов;
- селекторы псевдоэлементов.

При определении стиля можно использовать любую комбинацию селекторов.

В качестве селектора могут быть указаны теги. Например, ниже для тега параграфа <p> устанавливается желтый цвет фона, синий цвет текста и размер шрифта 18 пикселей.

```
p{
  background:yellow;
  font-size:18px;
  color:blue}
```

Атрибут id используется для выбора конкретного элемента, уникального в пределах страницы, поэтому селектор идентификатора используется, чтобы выбрать один уникальный элемент. Чтобы выбрать элемент с определенным идентификатором, надо перед идентификатором элемента указать хэш-символ (#). Id не может начинаться с цифры.

CSS-классы – это предопределенные стили на уровне тега. Класс стиля можно представлять как обычный стиль, имеющий имя. Это означает, что для каждого тега можно создать несколько стилей, каждый из которых будет иметь свое уникальное имя. При указании класса в селекторе после названия тега ставят точку, а затем указывают имя класса. Для применения стиля в атрибуте class соответствующего тега необходимо указать название класса.

```
.par2 {
    text-align: center;
    color: red;}
<p class="par2">Параграф определяется через class</p>
```

Можно также указать, что стиль относится не ко всем элементам, имеющим атрибут class с определенным значением, а только к конкретным тегам. Для этого начните с имени элемента, затем введите символ точки (.), а затем имя класса, например, в этом примере только те теги параграфа <p>, которые помечены как class="center" будут выровнены по центру. Комментарии используются для пояснения кода и могут помочь при редактировании исходного кода

спустя какое-то время. Комментарии игнорируются браузерами. Комментарии си подобные /**/.

```
div.par2 {
/* Меняем цвет шрифта*/
    color:aqua;
/* Разместим два блока,
К одному применим class="par2"*/}
…
<div class="par2">Блок определяется через class</div>
<div>Определяется блок</div>
```

Блок определяется через class

Определяется блок

В селекторах можно ссылаться на несколько классов. Применим стили:

```
.center {text-align: center;}
.large {font-size: 300%;}
```

Для кода:

```
<p class="center">Параграф выравнивается по центру</p>
<p class="large">Шрифт 300%</p>
<p class="center large">Параграф выравнивается по центру и шрифт
300%</p>
```

Универсальный селектор * позволяет выбрать все элементы веб-страницы вместо указания каждого тега по отдельности. Например, если нужно, чтобы все отображалось полужирным шрифтом, можно добавить следующий код:

```
a, p, img, h1, h2, h3, h4, h5 ...и т. д. тут надо перечислить все
теги документа { font-weight: bold; }
```

Использование символа * – более быстрый способ:

```
* {font-weight: bold;}
```

Довольно часто используют универсальный селектор * как способ отмены отступов вокруг блочных элементов. С помощью свойства margin можно добавить отступы вокруг элемента, а с помощью свойства padding – пространство между рамкой элемента и его содержимым. Для разных элементов браузеры автоматически формируют отступы различного размера, поэтому один из способов начать с чистого листа заключается в том, чтобы удалить все пространство вокруг элементов с определенным стилем:

```
* {
padding: 0;
margin: 0;
}
```

Кроме того, можно использовать универсальный селектор в составе селектора потомков, также называемого вложенным селектором. В этом случае применяете стиль ко всем элементам-потомкам, подчиненным определенному элементу веб-страницы. Например,

```
.banner * { font-weight: bold; }
```

выбирает все элементы внутри элемента, имеющего атрибут class="banner".

Часто используются также

Дочерние селекторы Селектор1 > Селектор2 { }

Соседние селекторы Селектор1 + Селектор2 { }
Селекторы атрибутов Селектор[атрибут] { }
Псевдоклассы :hover, :active, :target :focus, :first-child
Псевдоэлементы :after, :before, :first-letter, :first-line

дочерние селекторы – они **вложены** в родительский
соседние селекторы – они идут друг за другом **на одном уровне**, но только один последующий является соседом.

Пример.

```
*{margin: 0; }
P.menu{font-size:1.1em;}/* обратите внимание - нет пробела*/
UL#menu {margin: 0; padding: 0;}
P > EM { color: red } /*если является прямым потомком*/
P EM { color: red } /*допустим любой уровень вложенности*/
H2.sic + P { background: #ddd;} /*непосредственно первый абзац после заголовка */
a[target="_blank"] {font-weight: bold }
```

1.3.13. Группировка селекторов

Если одинаковый стиль назначается для нескольких элементов разметки, то их можно сгруппировать. Для этого эти теги должны быть перечислены через запятую в селекторе. Например: Три одинаковых стилевых правила для заголовка первого уровня, заголовка второго уровня и параграфа:

```
h1 {
    text-align: center;
    color: red;
}
h2 {
    text-align: center;
    color: red;
}
p {
    text-align: center;
    color: red;
}
```

Можно заменить одним правилом

```
h1, h2, p {
    text-align: center;
    color: red;
}
```

1.3.14. Единицы измерения длины

В CSS есть несколько различных единиц измерения длины. Длина представляет собой число с последующим указанием единицы длины, например 15px, 2em, 3% и т.д. Пробел между значением длины и единицей измерения не ставится. Есть единственное исключение: если значение равно 0, то единицу измерения можно не указывать. Для некоторых свойств CSS разрешена отрицательная длина.

В CSS существуют абсолютные и относительные единицы измерения длины.

Абсолютные единицы измерения используются в том случае, когда известны физические параметры устройства вывода. К абсолютным единицам измерения относятся:

in – дюйм (1 дюйм равен 2.54 сантиметра);

cm – сантиметр;

mm – миллиметр;

pt – пункт, (1 пункт равен 1/72 дюйма);

pc – пика, типографская мера (1 пика равна 12 pt или 4,23 мм);

px – пиксели (точки на экране компьютера).

Самой распространенной и широко используемой единицей измерения является пиксель.

Количество пикселей задаётся в настройках разрешения экрана, один px – это как раз один такой пиксель на экране. Все значения браузер в итоге пересчитает в пиксели.

Значения пикселей могут быть дробными, например размер можно задать в 16.5px. Например, есть элемент шириной в 100px, его нужно разделить на три части, отсюда появляются 33.3px. При окончательном отображении дробные пиксели округляются и становятся целыми.

Для мобильных устройств, у которых много пикселей на экране, но сам экран маленький, чтобы обеспечить читаемость, браузер автоматически применяет масштабирование.

В реальной жизни пиксель может быть разным, в зависимости от экрана. Однако в CSS под пикселем понимается абстрактный пиксель или точка на «стандартизованном экране», характеристики которого описаны в спецификации. Согласно спецификации в одном сантиметре содержится ровно 38 пикселей. Поэтому ни о каком соответствии cm реальному сантиметру говорить нельзя. Это полностью синтетическая и производная единица измерения. Из вышесказанных соображений mm, cm, pt, pc уже практически не используются.

Относительные единицы длины выражают отношение к чему-то другому, например, к размеру шрифта родительского элемента или к размеру области просмотра. Преимущество использования относительных единиц состоит в том, что при тщательном планировании можно сделать так, чтобы размер текста или других элементов масштабировался относительно всего остального на странице. Таблицы стилей, которые используют такой тип единиц, намного легче перенастраиваются с одного типа устройств на другой. Например, с экрана компьютера на лазерный принтер и т. д. К относительным единицам измерения относятся:

ch – относительно ширины символа "0" шрифта элемента;

em – относительно размера шрифта родительского элемента, 2em означает в 2 раза больше размера текущего шрифта;

ex – относительно высоты буквы x (икс) в латинском алфавите текущего шрифта (редко используется);

rem – относительно размера шрифта корневого элемента <html>;

vw – относительно 1% ширины области просмотра. Область просмотра (viewport) = размер окна браузера. Если окно просмотра имеет ширину 50 см, 1vw = 0,5 см;

vh – относительно 1% высоты области просмотра (viewport);

vmin – относительно 1% от минимального размера области просмотра (наименьшее из vw и vh);

vmax – относительно 1% от максимального размера области просмотра (наибольшее из vw и vh);

% – относительно размера родительского элемента.

Приведем пример (рисунок 29).

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Документ без названия</title>
    <style>
      .box { border: solid thick #461EA4;
             background: #ABD6E7;
             padding: 10px;  margin: 10px;}
            .wrapper {width: 50%;}
            .px {width: 200px;}
            .vw {width: 25vw;}
            .em {font-size: 2em; width: 33%;}
    </style>
  </head>
  <body>
    <div class="box wrapper">Это внешний родительский
      элемент, его ширина половина области просмотра окна
      браузера.
      <div class="box px">Это 1-й дочерний  элемент.
        Его ширина 200px
      </div>
      <div class="box vw">Это 2-й дочерний  элемент. Его
        ширина 25vw (25% ширины области просмотра)
      </div>
      <div class="box em">Это 3-й дочерний  элемент.
       Его ширина  33% (от ширины родительского элемента),
       а размер шрифта в 2 раза больше текущего шрифта
      </div>
    </div>
  </body>
</html>
```

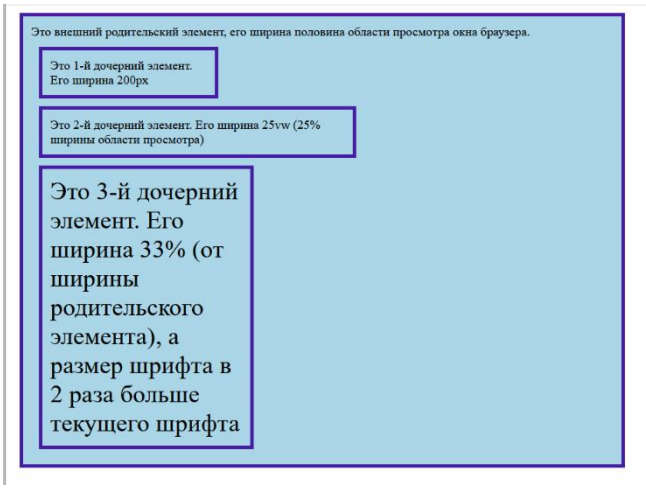


Рисунок 29 – задание ширины блока.

В данном примере во внешнем блоке, оборачивающем все остальные элементы установлена ширина width: 50%, т. е. половина области просмотра окна браузера.

Во внешний родительский блок вложен первый дочерний элемент – тоже блок. Его ширина 200px.

Для второго вложенного блока установлено стилевое правило width: 25vw, т. е. . его ширина составляет 25% ширины области просмотра окна браузера.

В третьем дочернем блоке установлена ширина width: 33%, т.е. треть от ширины области просмотра браузера, а размер шрифта в 2 раза больше текущего шрифта.

1em обозначает текущий размер шрифта. Можно брать любые пропорции от текущего шрифта: 2em, 0.5em и т.п. Размеры в em – относительные, они определяются по текущему контексту.

Приведем пример использования em.

```
<body style="font-size:20px">
  <p>Многим нравятся гарики.</p>
  <div style="font-size:1.5em">
    Бывает – проснешься, как птица, крылатой пружиной на
    взводе, и хочется жить и трудиться;
    <span style="font-size:1.5em">но к завтраку это
    проходит.</span>
  </div>
</body>
```

Многим нравятся гарики.

Бывает – проснешься, как птица, крылатой пружиной на взводе, и хочется жить и трудиться; но к завтраку это проходит.

Так как значение в em высчитывается относительно текущего шрифта, то текст вложенный в 1.5 раза больше, чем текст в параграфе. Фраза в еще в 1.5 раза больше, чем .

Поскольку em – единица относительная, размеры, заданные в em, будут уменьшаться или увеличиваться вместе с размером базового шрифта. С учетом того, что размер шрифта обычно определяется в родителе, и может быть изменен ровно в одном месте, это бывает очень удобно.

Vw, vh, vmin, vmax были созданы для поддержки мобильных устройств. Их основное преимущество в том, что любые размеры, которые в них заданы, автоматически масштабируются при изменении размеров окна.

1.3.15. Вычисляемые значения. Функция calc()

Функция calc() используется для указания вычисляемого значения свойств CSS, которые в качестве значения используют какое-либо число. В первую очередь calc можно применять для вычисления размеров, углов, времени. Причем функция позволяет смешивать различные единицы измерений, например ширину блока div можно задать следующим образом:

```
div {width: calc (100% – 40px);}
```

Ширина блока будет на 40 пикселей меньше размера родительского блока.

Другой пример. На странице есть два блока. Ширина первого блока на 100px больше, чем половина окна браузера. В этом блоке есть фоновый рисунок. Он смещен на 20px от правого края блока, при этом сам блок «резиновый»: calc(100% – 20px). Это пример того как при адаптивной верстке встраивать фоновые изображения. Ширина второго блока на 20px меньше, чем 80% доступной ширины: calc(80% – 20px).

1.3.16. Браузерные префиксы

Веб-разработчики перед некоторыми css-свойствами указывают браузерные префиксы: -webkit-, -moz-, -ms- и др. Например,

```
* {
    -moz-box-sizing: border-box; /* Для Firefox */
    -webkit-box-sizing: border-box; /* Для Opera, Safari и
    Chrome */
    -ms-box-sizing: border-box; /* Для IE */}
```

На данный момент используются следующие префиксы:

-webkit- для браузеров Chrome, Safari, Opera;

-moz- для браузера Mozilla Firefox;

-ms- для браузера Internet Explorer.

Если перед названием свойства стоит некоторый префикс, то это означает, что данное свойство реализовано и будет применяться исключительно в указанном браузере. Все остальные браузеры данное свойство будут игнорировать, т.к. для них данный префикс неизвестен. В примере выше в Mozilla Firefox будет использоваться свойство -moz-box-sizing, в Opera, Safari и Chrome – -webkit-box-sizing, а Internet Explorer -ms-box-sizing.

Префиксы возникают:

При включении в браузер экспериментальных свойств CSS, которые стандартом еще не утверждены. Таким образом, производители браузеров произво-

дят тестирование и вносят предложения перед утверждением свойств CSS в стандарте.

При создании собственных свойств, которые не входят в стандарт CSS, но возможно появятся в нем через некоторое время.

Для решения проблем с кроссбраузерностью.

Когда экспериментальное свойство утверждено в стандарте и прошло тестирование в браузере, у него обычно убирается префикс.

Рассмотрим в качестве примере следующий код:

```
* {
    -moz-box-sizing: border-box; /* Для Firefox */
    -webkit-box-sizing: border-box; /* Для Opera, Safari
     и Chrome */
     box-sizing: border-box;}
```

Данный код применяет свойства CSS, которые изменяют алгоритм расчета ширины и высоты для всех элементов веб-страницы. Первое CSS свойство -moz-box-sizing со значением border-box предназначено для браузеров, использующих движок Gecko (Mozilla Firefox). Второе CSS свойство -webkit-box-sizing со значением border-box предназначено для браузеров, использующих движок webkit (Safari) или blink (Chrome, Opera, Яндекс.Браузер). Последнее CSS свойство предназначено для браузеров, в которых это свойство уже протестировано и внедрено в соответствии со стандартом.

При использовании префиксов для свойств CSS, необходимо их размещать до свойства CSS без префикса. Это важно потому, что если когда-то в браузере будет реализовано оригинальное свойство (без префикса), то будет использоваться именно оно (т.к. оно располагается последним), а не его экспериментальная версия.

Проверить поддержку определенного свойства в браузере можно, например, на сайте https://caniuse.com (смотри рисунок 30).

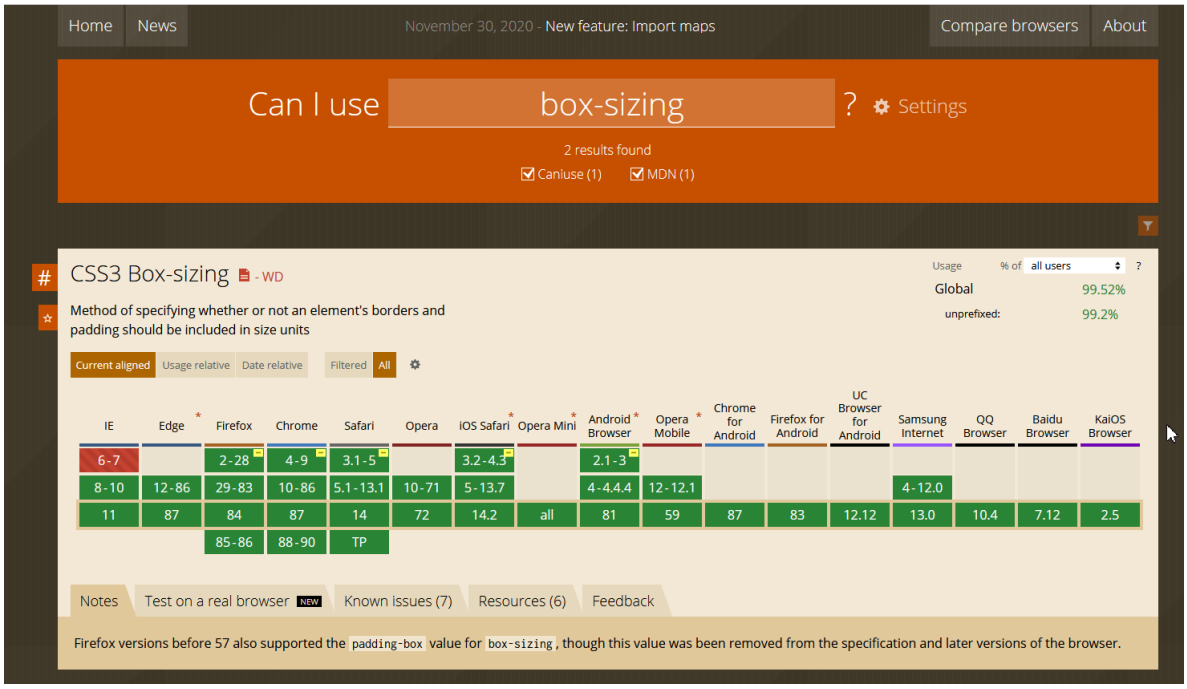


Рисунок 30 – Проверка поддержки свойств браузером.

Кроме этого на сайте показывается количество пользователей в процентах, которые пользуется этой версией браузера.

На сайте https://caniuse.com браузеры отмечаются различными цветами, в зависимости от того в каком состоянии находится поддержка определенных свойств или тегов:

Красный прямоугольник – браузер, в котором данное свойство не реализовано.

Зеленый прямоугольник с дефисом, расположенным в правом верхнем углу – браузер, в котором данное свойство используется через префикс.

Светло-зеленый прямоугольник – браузер, в котором данное свойство реализовано частично.

Зеленый прямоугольник – браузер, в котором данное свойство реализовано в соответствии со стандартом.

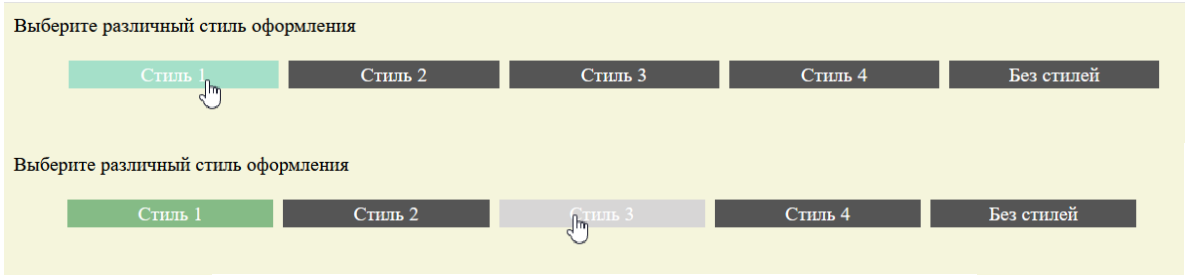
1.3.17. Подключение альтернативных стилей

Приведем пример подключения альтернативных стилей.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
/* Stylesheet 1: */
    <style>
      body {background-color: beige;}
      #s0  {background-color: rgba(133,187,134,1.00);}
      #s0:hover {
            background-color:rgba(165,224,201,1.00) ;}
    </style>
>/* Stylesheet 2: */
    <style
      body {background-color: aqua;}
```

```
      #s1  {background-color: rgba(133,187,134,1.00);}
      #s1:hover {
            background-color:rgba(165,224,201,1.00) ;}
    </style>
/* Stylesheet 3: */
    <style>
      Body {background-color: blueviolet;}
      #s2  {background-color: rgba(133,187,134,1.00);}
      #s2:hover {
            background-color:rgba(165,224,201,1.00) ;}
    </style>
/* Stylesheet 4: */
    <style>
      body {background-color: aquamarine;}
      #s3  {background-color: rgba(133,187,134,1.00);}
      #s3:hover {
            background-color:rgba(165,224,201,1.00) ;}
    </style>
    <style>
      .menuitem {
            width: 165px;
            float: left;
            background-color: #555555;
            color: #ffffff;
            list-style-type: none;
            margin: 4px;
            padding: 2px;
            text-align: center;
            cursor: pointer;}
      .menuitem:hover {
            background-color:rgba(215,214,214,1.00);}
    </style>
  </head>
  <body>
    Выберите различный стиль оформления
    <ul >
     <li  class="menuitem" id="s0"
      onclick="reStyle(0)">Стиль 1</li>
     <li class="menuitem" id="s1"
      onclick="reStyle(1)">Стиль 2</li>
     <li class="menuitem" id="s2"
      onclick="reStyle(2)">Стиль 3</li>
     <li class="menuitem" id="s3"
      onclick="reStyle(3)">Стиль 4</li>
     <li class="menuitem" id="s4"
      onclick="noStyles();
      document.styleSheets[4].disabled = true;">
      Без стилей</li>
    </ul>
    <script>
      function noStyles() {
            document.styleSheets[0].disabled = true;
            document.styleSheets[1].disabled = true;
```

```
            document.styleSheets[2].disabled = true;
            document.styleSheets[3].disabled = true;
            document.styleSheets[4].disabled = false;   }
      function reStyle(n) {
            noStyles();
            document.styleSheets[n].disabled = false;
         /*только n-ую таблицу стилей делаем доступной*/}
            reStyle(n);
    </script>
  </body>
</html>
```



Выберите различный стиль оформления

- Стиль 1
- Стиль 2
- Стиль 3
- Стиль 4
- Без стилей

Рисунок 30 – стили отличаются только фоном.

1.3.18. Теги блочные и строчные.

По умолчанию браузер большинство тегов отображает как блочные (из часто используемых, только <img>, , <a> не являются блочными). Это значит, что они занимают всю доступную ширину, имеют до и после себя отбивку, высота элемента определяется его содержимым, и он всегда начинается с новой строки. Естественно, что с помощью стилей для любго элемента страницы можно изменить его display: block | inline | flex | inline-block | inline-table | list-item | none | run-in | table | table-caption | table-cell | table-column-group | table-column | table-footer-group | table-header-group | table-row | table-row-group

Для блочных элементов характерны следующие особенности.

- Блоки располагаются по вертикали друг под другом.
- На прилегающих сторонах элементов действует эффект схлопывания отступов.
- Запрещено вставлять блочный элемент внутрь строчного. Например, <a><h1>Заголовок</h1> не пройдёт валидацию, правильно вложить теги наоборот — <h1><a>Заголовок</h1>.
- По ширине блочные элементы занимают всё допустимое пространство.

- Если задана ширина контента (свойство width), то ширина блока складывается из значений width, полей, границ, отступов слева и справа.
- Высота блочного элемента вычисляется браузером автоматически, исходя из содержимого блока.
- Если задана высота контента (свойство height), то высота блока складывается из значения height, полей, границ, отступов сверху и снизу. При превышении указанной высоты контент отображается поверх блока.
- На блочные элементы не действуют свойства, предназначенные для строчных элементов, вроде vertical-align.
- Текст по умолчанию выравнивается по левому краю.

Для строчных элементов характерны следующие особенности.

- Внутрь строчных элементов допустимо помещать текст или другие строчные элементы. Вставлять блочные элементы внутрь строчных запрещено.
- Эффект схлопывания отступов не действует.
- Свойства, связанные с размерами (width, height) не применимы.
- Ширина равна содержимому плюс значения отступов, полей и границ.
- Несколько строчных элементов идущих подряд располагаются на одной строке и переносятся на другую строку при необходимости.
- Можно выравнивать по вертикали с помощью свойства vertical-align.

inline-flex Элемент ведёт себя как строчный и выкладывает содержимое согласно флекс-модели.

flex Элемент ведёт себя как блочный и выкладывает содержимое согласно флекс-модели.

1.3.19. Стандартная блочная модель

В CSS используется модель представления частей html-документа в виде прямоугольных блоков. Каждый блок имеет информативную область, в которой заключено содержимое породившего его элемента (например, текст, изображение и т.п.), а также может иметь области, отведенные для оформления полей (padding), рамок (border) и внешних отступов (margin).

Чтобы правильно установить ширину и высоту элемента необходимо знать, как работает блочная модель (Рисунок 31).

Размеры окантовки существенно влияют не только на внешний вид блока, но и на размер пространства, которое блок будет занимать в браузере. Ширина информационной области, как правило, задаётся свойством width, а высота через height. Поле слева и справа от контента до рамки занимают padding-left и padding-right соответственно. Рамка также имеет ширину. margin-left и margin-right – отступы слева и справа от блока.

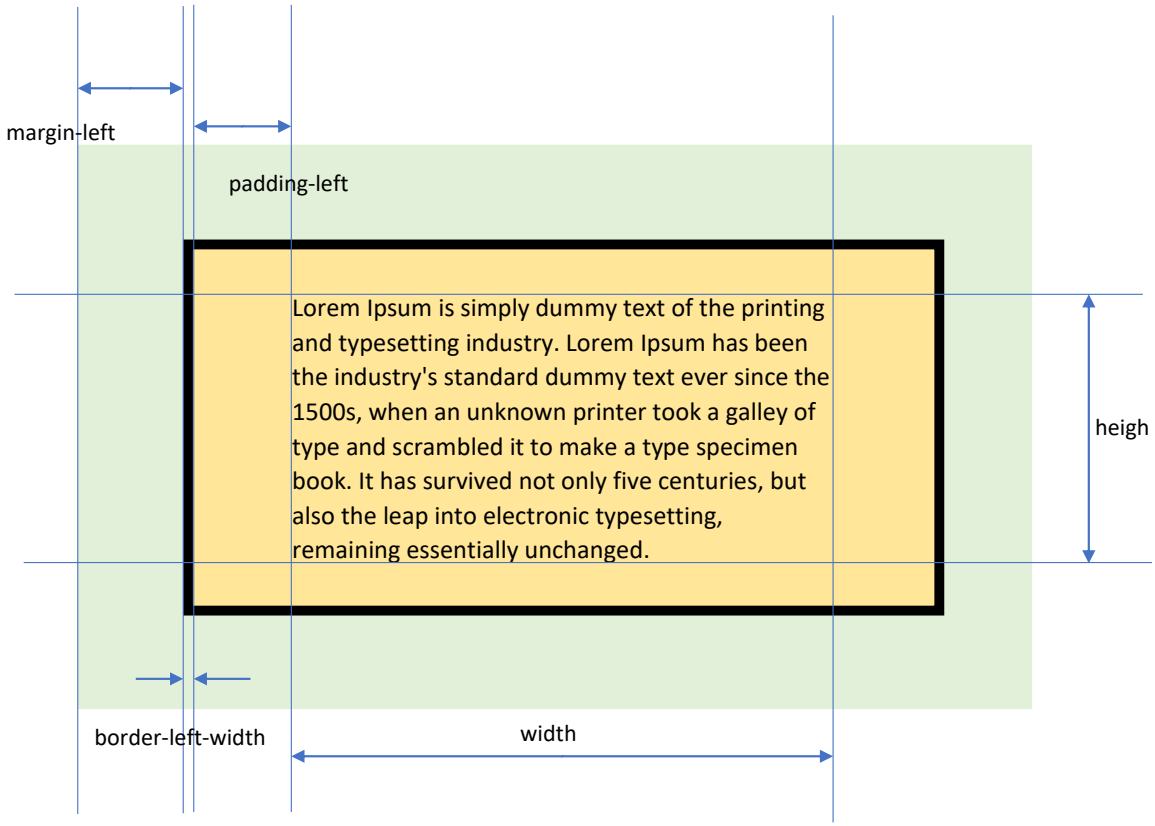


Рисунок 31 – стандартная блочная модель.

Ширина места, которое будет занимать блок в макете рассчитывается следующим образом: Общая ширина элемента = ширина (ширина контента) + левый отступ + правый отступ + левая граница + правая граница + левое поле + правое поле или width + padding-left + padding-right + border-left + border-right + margin-left + margin-right.

Например, если есть стиль

```
.box{
      width: 1110px;
      padding-left: 55px;
      padding-right: 54px;
      margin: 40x;
      border: solid 2px red;
}
```

Согласно этому правилу элемент с классом box по ширине будет занимать 1110 + 55 + 54 + 2 + 2 + 40 + 40 = 1303px.

Общая высота элемента рассчитывается аналогично: Общая высота элемента = высота + верхний отступ + нижний отступ + верхняя граница + нижняя граница + верхнее поле + нижнее поле.

1.3.20. Альтернативная блочная модель

При использовании альтернативной модели, любая ширина – это ширина видимой части элемента на странице, поэтому ширина области содержимого будет равна общей ширине минус сумму ширины левой и правой границы и минус сумму левого и правого внутренних полей.

Ширина контента = ширина (общая ширина элемента) – (левая граница + правая граница) – (левое внутреннее поле + правое внутреннее поле).

1.3.21. Проблемы стандартной блочной модели

Стандартная блочная модель, когда свойство width задаёт ширину контента, не слишком удобна. Постоянно приходится заниматься вычислениями, когда требуется задать определенную ширину блока. Также начинаются проблемы при сочетании разных единиц измерения, в частности, процентов и пикселов. Предположим, что ширина контента задана как 90%, если сюда приплюсовать поля и рамки, заданные в пикселах, то нельзя вычислить суммарную ширину блока, поскольку проценты напрямую в пикселы не переводятся. В итоге может получиться так, что общая ширина блока превысит ширину вебстраницы, что приведет к появлению горизонтальной полосы прокрутки. В качестве выхода из подобной ситуации можно использовать вложенные блоки или изменить принцип построения блочной модели.

Рассмотрим оба варианта.

При использовании вложенных блоков, для нужного блока создаётся блок-обертка. Для внешнего блочного элемента задаётся только необходимая ширина. Для вложенного блока задаются все остальные свойства – поля, рамки и отступы. Поскольку по умолчанию ширина блока равна доступной ширине родителя, получится, что блоки в каком-то смысле накладываются друг на друга, при этом фактическая ширина такого комбинированного элемента будет четко задана.

Можно изменить принцип построения блочной модели и использовать альтернативную блочную модель.

В CSS3 есть свойство box-sizing. По умолчанию значение свойства равно content-box, в этом случае width и height блока включает только контент, а border и padding не включают. При значении box-sizing равном border-box ширина начинает включать поля и рамки, но не отступы. Таким образом, подключая box-sizing со значением border-box к своему стилю, можно задавать ширину в процентах и спокойно указывать border и padding, не боясь, что изменится ширина блока.

Это прекрасное решение, но оно имеет существенный недостаток: не все браузеры его поддерживают.

Допустим нам надо сверстать блок, который будет занимать на странице по ширине 1219px. Расстояние от левого края блока до текста должно быть 95px, о от правого края блока до текста – 94px. В приведенном ниже примере (рисунок 32) даётся два варианта верстки: с использованием стандартной блочной верстки и с использованием альтернативной блочной модели.

```
<!doctype html>
<html>
  <head>
    <style>
/*Вариант 1*/
      .box{
```

```
        padding-left: 55px;
        padding-right: 54px;
        margin: 40px;
        border: solid 2px red;
        background-color:coral;  }
      .wrapper{
        width: 1219px;
        background-color: bisque;}
/*Вариант 2*/
      .boxsizing{
        -moz-box-sizing: border-box; /* Для Firefox */
        -webkit-box-sizing: border-box; /* Для Opera,
        Safari и Chrome */
        box-sizing: border-box; /* Для IE */
        width: 1219px;
        padding-left: 95px;
        padding-right: 94px;
        margin: 0px;
        border: solid 2px red;
        background-color: darkturquoise;}
    </style>
  </head>
  <body>
<!--Вариант 1-->
    <div class="wrapper">
      <div class="box">
        Lorem Ipsum is simply dummy text of the printing
        and typesetting industry. Lorem Ipsum has been the
        industry's standard dummy text ever since the
        1500s, when an unknown printer took a galley of
        type and scrambled it to make a type specimen
        book.
      </div>
    </div>
<!--Вариант 2-->
    <div class="boxsizing">
      Lorem Ipsum is simply dummy text of the printing
      and typesetting industry. Lorem Ipsum has been the
      industry's standard dummy text ever since the 1500s,
      when an unknown printer took a galley of type and
      scrambled it to make a type specimen book.
    </div>
  </body>
</html>
```

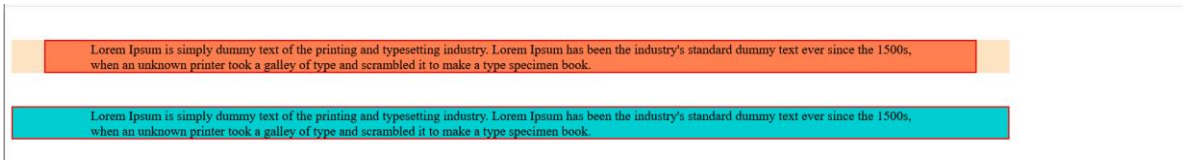


Рисунок 32 – при верстке блоки занимают одинаковое место.

1.3.22. Размеры блока

Значение ширины и высоты блока могут указываться в абсолютных величинах, например пикселях, или в относительных, например, в процентах. При использовании процентной записи ширина элемента вычисляется в зависимости от ширины родительского элемента. Если родительский элемент явно не указан, то в его качестве берется окно браузера.

Свойства блока width и height

Свойство width и height не применяются к строчным элементам display: inline. Ширина и высота содержимого встроенных блоков определяется шириной отображаемого содержимого внутри них. Отрицательные значения не допускаются.

Свойства блока min-width и max-width, min-height и max-height

Свойства min-width и max-width позволяют ограничивать ширину содержимого до определенного значения. Предельные значения не могут быть отрицательными. Для min-width значение по умолчанию 0, для max-width – none, что означает, что ограничения на ширину блока отсутствуют.

Аналогично, свойства min-height и max-height ограничивают высоту элементов определенным диапазоном.

Свойства блока padding, border, margin

Свойство margin описывает ширину отступа между рамкой блока и границей элемента, в который помещен блок. Свойство border определяет свойства рамки блока. Свойство padding описывает поля или отступы от рамки блока до его содержимого.

Каждое из свойств padding, border, margin имеют соответствующие частные свойства: *-top, *-bottom, *-left, *-right, где вместо звездочки используется название одного из трех свойств. Каждое из частных свойств может быть задано отдельно. Одновременно в базовом свойстве может быть задано от одного до четырех значений частных свойств. Когда указано одно значение, оно применяется ко всем четырем сторонам. Например, следующие объявления эквивалентны:

```
p {margin:7px;}
p {
   margin-top:7px;
   margin-right:7px;
   margin-bottom:7px;
   margin-left:7px;
}
```

Когда указаны два значения, первый отступ применяется сверху и снизу, второй справа и слева. Например, следующие объявления эквивалентны:

```
p {margin:7px 10px;}
p {
   margin-top:7px;
   margin-right:10px;
   margin-bottom:7px;
```

```
    margin-left:10px;
}
```

Когда заданы три значения, первый отступ применяется сверху , второй слева и справа , третий – снизу.

```
p {margin:7px 10px 14 px}
p {
    margin-top:7px;
    margin-right:10px;
    margin-bottom:14px;
    margin-left:10px;
}
```

Если указано четыре значения отступа, то они применяются сверху , справа, снизу и слева.

```
p {margin:7px 10px 14 px 5px;}
p {
    margin-top:7px;
    margin-right:10px;
    margin-bottom:14px;
    margin-left:5px;
}
```

Запомнить порядок применения значений просто: они присваиваются по часовой стрелке, начиная сверху. Например:

```
p {
    border-style: solid;
    border-top-width:2px;
    border-right-width:4px;
    border-bottom-width:6px;
    border-left-width:8px;
}
```

Так как поля и отступы элемента не являются обязательными, по умолчанию их значение равно нулю. Тем не менее, браузеры добавляют этим свойствам положительные значения по умолчанию на основе своих таблиц стилей. Очистить стили браузеров для всех элементов можно при помощи универсального селектора:

```
* { margin: 0;
    padding: 0;
}
```

Свойства блока padding

Поля или padding представляет собой пространство между краем области содержимого и рамкой элемента. Поля прозрачные и фон элемента по умолчанию закрашивает его поля и пространство под его рамкой. Значения полей определяют толщину их области. Сокращенное свойство padding задаёт поля для всех четырех сторон, а подсвойства устанавливают только их соответствующие стороны.

Демонстрация использования сокращенной записи свойства padding.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
```

```
    <style>
      div {border: 1px solid black;
           background-color: lightblue;
           }
      .ex1 {padding: 25px 50px 75px 100px;}
      .ex2 {padding: 25px 50px 75px;}
      .ex3 {padding: 25px 50px;}
      .ex4 {padding: 25px; }
    </style>
  </head>
  <body>
    <h2>Использование сокращенной записи для определения
     полей элемента
    </h2>
    <div class="ex1">Верхнее поле 25px, правое поле 50px,
     нижнее поле 75px, а левое поле 100px.
    </div>
    <br>
    <div class="ex2">Верхнее поле 25px, правое и левое
      поля 50px, нижнее поле 75px.
    </div>
    <br>
    <div class="ex3">Верхнее поле и нижнее поля 25px,
      правое и левое поля 50px.
    </div>
    <br>
    <div class="ex4">Все поля по 25px. Все поля по 25px.
      Все поля по 25px. Все поля по 25px.
    </div>
    <br>
  </body>
</html>
```

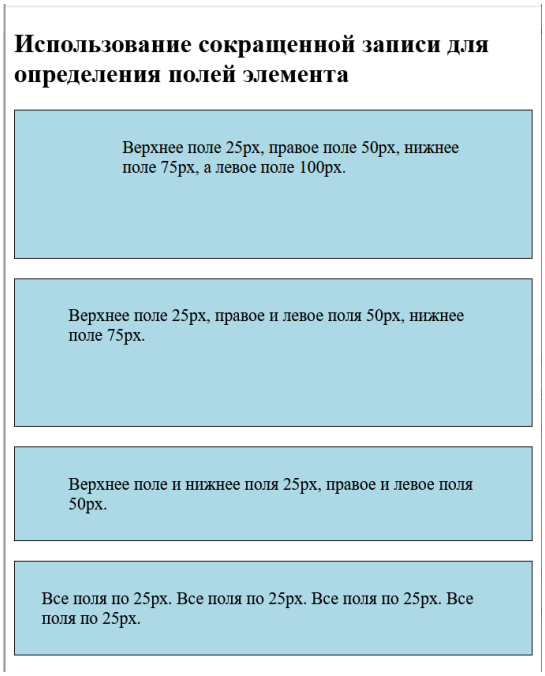


Рисунок 33 – свойства padding.

1.3.23. Свойства блока margin

Свойство margin используются для создания пустого пространства вокруг элементов за пределами определенных границ. Внешние отступы прозрачные. Применяются ко всем элементам, кроме внутренних элементов таблицы.

Иногда необходимо центрировать элемент в его контейнере в горизонтальном направлении. В этом случае можно установить свойство margin в auto. Элемент будет занимать указанную ширину, а остальное пространство будет разделено поровну между левым и правым полями. В вертикальном направлении это не работает (рисунок 34).

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
      div {
           width:300px;
           margin: auto;
           border: 1px solid red;
           }
    </style>
  </head>
  <body>
    <h2>Использование значения auto для margin</h2>
    <p>Можно установить свойство margin в auto для
     горизонтального центрирования элемента внутри
     контейнера. Элемент будет занимать указанную ширину,
     а остальное пространство будет разделено поровну
     между левым и правым полями:
    </p>
    <div>
     Этот div будет выравнен по центру. потому что у него
     margin: auto;
    </div>
  </body>
</html>
```

Рисунок 34 – свойства padding.

Свойство margin автоматически не наследуется. Следующий пример на рисунке 35 показывает как можно унаследовать левый отступ от родительского элемента:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
```

```
      div.container {
           margin-left: 100px;
           border: 1px solid red;}
      p.one {
           margin-left: inherit;
           border: 1px solid;}
    </style>
  </head>
  <body>
    <h2>Использование наследуемых значений</h2>
    <p>Пусть левый отступ margin наследуется от
      родительского элемента:
    </p>
    <div class="container">
      <p class="one">
        Этот параграф наследует от родительского тега div
        левый отступ margin.
      </p>
    </div>
  </body>
 </html>
```

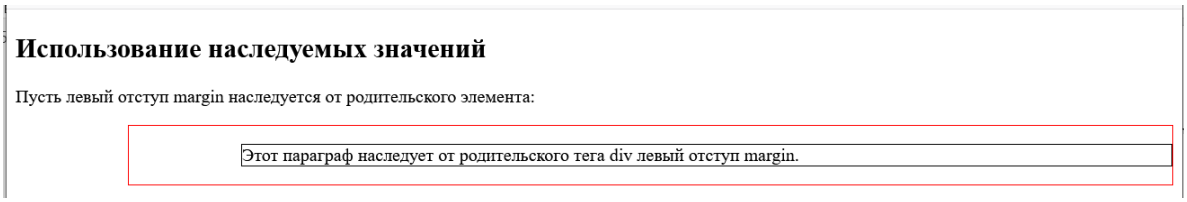


Рисунок 35 – свойство margin.

Значение свойства margin может быть как положительным, так и отрицательным. Отрицательное значение для внешнего отступа может привести к перекрытию элементов страницы. Независимо от того, используется стандартная или альтернативная блочные модели, margin всегда добавляется после расчета размера видимой части блока.

Особенности вертикального суммирования margin

Верхние и нижние отступы элементов сжимаются в один margin, который равен наибольшему из двух полей (рисунок 36).

```
 <!doctype html>
 <html>
   <head>
     <meta charset="utf-8">
     <style>
       div {
            margin: 0;
            width: 100px;
            border: 1px solid red;
            }
       .one {margin-bottom: 40px;}
       .two {margin-top: 10px;}
     </style>
   </head>
   <body>
     <div class="one">one</div>
```

```
    <div class="two">two</div>
  </body>
</html>
```

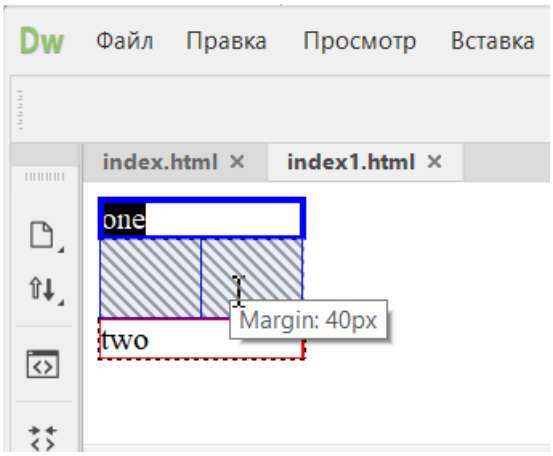


Рисунок 36.

В данном примере есть два блока div фиксированной ширины. Для наглядности у блоков задана граница. Первый блок имеет отступ снизу 40px. У второго блока верхний отступ 10px. Но как видно из картинки, реально расстояние между первым и вторым блоками 40px. Т.е. вертикальные отступы двух элементов уровня блока перекрываются. При этом ширина общего отступа равна ширине большего из исходных.

Если один из отступов отрицательный (рисунок 37), и величина прилегающих отступов элементов x и y, то происходит складывание отступов по правилам математики: x + (-y) = x – y.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
      div {
          margin: 0;
          width: 100px;
          border: 1px solid red;}
   .one {margin-bottom: 40px;}
   .two {margin-top: -15px;}
    </style>
  </head>
  <body>
    <div class="one">one</div>
    <div class="two">two</div>
  </body>
</html>
```

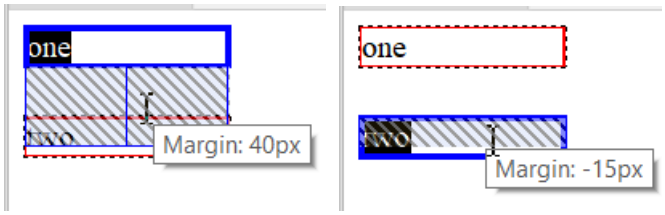


Рисунок 37.

Если оба отступа отрицательны (рисунок 38), то из двух значений выбирается наибольшее по модулю, оно же и выступает в качестве отрицательного отступа между элементами. Так, если отступы равны -15px и -7px, то итоговое значение будет -15px.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
      div {
            margin: 0;
            width: 100px;
            border: 1px solid red;
           }
      .one {margin-bottom: -15px;}
      .two {margin-top: -7px;}
    </style>
  </head>
  <body>
    <div class="one">one one one one one</div>
    <div class="two">two two two two two two two</div>
  </body>
</html>
```

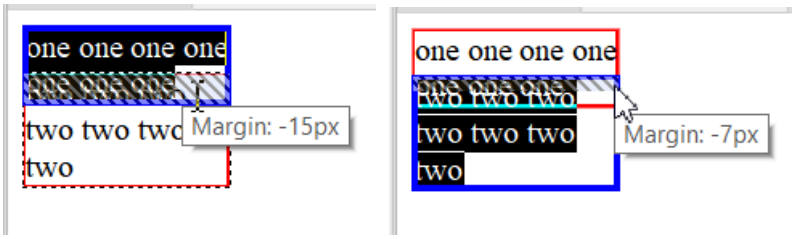


Рисунок 38.

Горизонтальное суммирование margin

Горизонтальные отступы между элементами просто складываются (рисунок 39). Например, горизонтальный отступ между двумя элементами с отступами 15px будет равен 30px.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
      div {
           margin: 0;
           width: 100px;
           float: left;
           border: 1px solid red;
          }
      .one {margin-right: 15px;}
      .two {margin-left: 15px;}
    </style>
  </head>
  <body>
    <div class="one">one one one one one one one</div>
```

```
    <div class="two">two two two two two two two</div>
  </body>
</html>
```

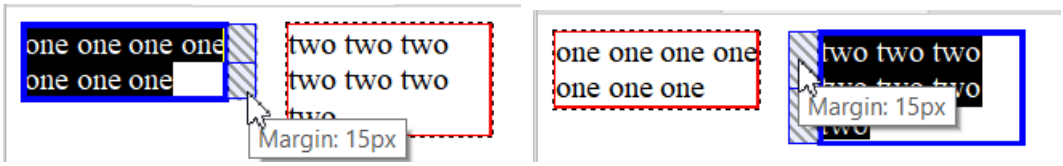


Рисунок 39.

Вертикальное схлопывание внешних полей, когда элементы связаны родительской связью

Если элементы связаны родительской связью, то схлопывание работает немного сложнее. Если внутри одного блока расположить другой блок и задать ему margin-top, то внутренний блок прижмется к верхнему краю родительского, а у родительского элемента появится отступ сверху, т.е. внутренний блок как бы «выпадет» из родительского блока. Если у родительского элемента также был задан верхний отступ, то выберется наибольшее из значений.

Для рассмотрения данной ситуации представим, что имеется два блока: блок-родитель и дочерний блок. Для того, чтобы сработало схлопывание, нужно что бы блок-родитель не имел ни полей, ни границ (их значения были нулевые). Таким образом, внешние поля этих элементов соприкасаются. Отметим, что нулевые значения внутренних полей и границ для родителей – обязательное условие срабатывания схлопывания.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
      body {
            background: url("разлинованные клетки.png")
                        repeat;}
      div {
            width:200px;
            text-align:center;
          }
      .parent {
            /*Полей у родителя быть не должно*/
            padding:0px;
            /*Границ у родителя быть не должно */
            border:0px solid;
            /* Этот отступ схлопнулся с дочерним */
            margin-top: 40px;
            background-color:aqua;
            }
      .daughter {
            width:186px;
            /* Актуальное значения отступа */
            margin-top:60px;
            /* У дочернего может быть граница*/
            border:4px solid red;
            /* У дочернего блока внутренние отступы могут
```

```
                быть не равны 0 */
               padding:3px;
               background-color:yellow;
               }
    </style>
  </head>
  <body>
    <div class="parent">
      <div class="daughter">Дочерний</div>
        Родительский
    </div>
  </body>
</html>
```

Для наглядности в качестве фона установим изображение с разлинованными клетками (рисунок 40). Размер одной клетки равен 20 пикселей, поэтому без схлопывания отступ между элементами был бы 60px +40px = 100px, но из-за применения данного эффекта он равен максимальному значению одного из элементов (дочернего), в данному случае – 60 пикселям (3 клетки). Отступ дочернего блока заменяет верхний отступ для своего родителя.

Рисунок 40.

Если, например, добавить границу к родительскому блоку, то схлопывания не произойдет (рисунок 41).

```
.parent {
        padding:0px;/* Полей у родителя быть не должно */
        border:2px solid;
        margin-top: 40px;
        background-color:aqua;
        }
```

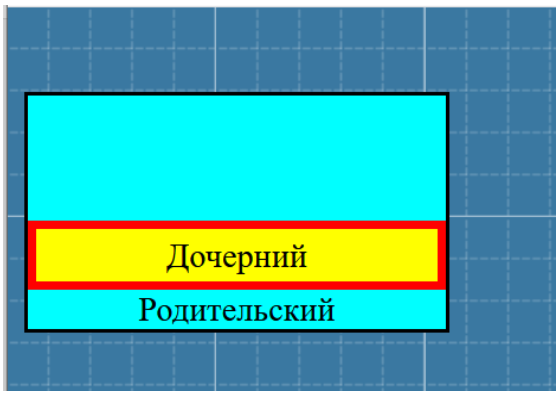


Рисунок 41.

Поэтому, чтобы избавиться от схлопывания, достаточно задать родительскому элементу padding-top или добавить border-top: 1px solid transparent.

1.3.24. Рамки элемента

Свойство border задаёт рамку элемента и представляет собой одну или несколько линий, окружающих содержимое элемента и его поля padding. Стиль рамки задаётся с помощью трех свойств: стиль, цвет и ширина.

border-style

Свойство border-style устанавливает тип рамки одновременно на всех сторонах элемента разметки или индивидуально для каждой стороны. Способ изменения типа рамки зависит от числа аргументов. Может быть задано от одного до четырех значений. По умолчанию рамки всегда отрисовываются поверх фона элемента. Если не задан стиль рамки, то без этого свойства рамка не будет отображаться вообще (рисунок 42).

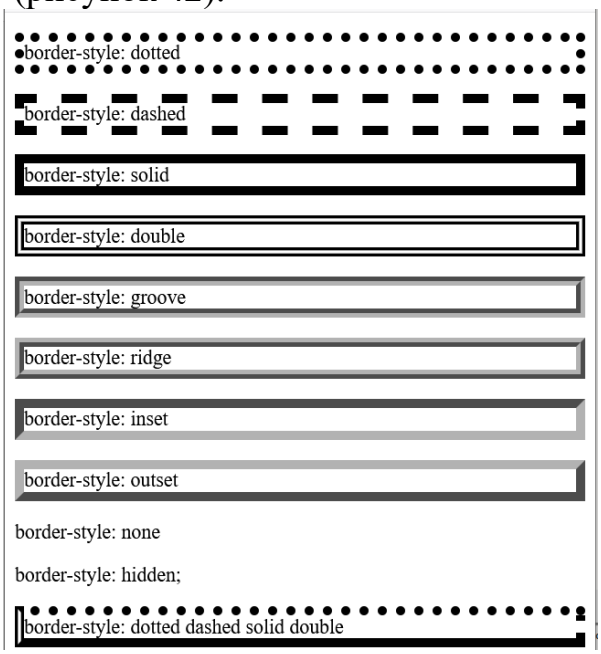


Рисунок 42.

Свойство border-style может принимать следующие значения:
none – граница не отображается;
hidden – определяет скрытую границу;
dotted – определяет пунктирную границу (точками);
dashed – определяет пунктирную границу (прерывистой линией);
solid – определяет сплошную границу;
double – определяет сплошную двойную границу;
groove – определяет трехмерную рифленую границу (вогнутая рамка);
ridge – определяет трехмерную рифленую границу (выпуклая рамка);
inset – понижение содержимого блока по отношению к границе;
outset – повышение содержимого блока по отношению к границе;
initial – устанавливает в значение по умолчанию, т.е. none;
inherit – использует значение свойства родительского элемента.

В разных браузерах вид отдельных стилей границы может отличаться. Например, при применении стиля dotted к рамке точка может изображаться в виде ромбика, в виде квадратика или кружочка.

Для управления стилем рамки только одной стороны элемента разметки можно воспользоваться свойствами border-top-style, border-right-style, border-bottom-style и border-left-style.

Рамки элемента border-width

Свойство border-width задаёт ширину границы одновременно на всех сторонах элемента или индивидуально для каждой стороны. Способ изменения ширины зависит от числа аргументов. Может быть задано от одного до четырех значений. Ширина границы может быть задана числовым значением. Также доступны три предопределенных значения: thin (2px), medium (4px), thick (6px), рисующие тонкую, среднюю и толстую линии соответственно. Значение по умолчанию medium.

Для управления шириной рамки только одной стороны (рисунок 43) элемента разметки можно воспользоваться свойствами border-top-width, border-right-width, border-bottom-width и border-left-width.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
     div {
             border-style:solid;
             border-width:0px;
         }
     div#s1 {border-width:thick;}
     div#s2 {border-width:10px;}
     div#s3{ border-width:2px 4px 6px 8px;}
     div#s4 {
             border-left-width:10px;
             border-bottom-width:1px;
            }
    </style>
  </head>
  <body>
    <div>Текст можно не обрамлять:
      border-width:0px
     </div>
    <br>
    <div id="s1"> Текст можно обрамить толстой линией,
      толщину линии задать предопределенным значением:
      border-width:thick
    </div>
    <br>
    <div id="s2"> Текст можно обрамлять еще более
      толстой линией, толщину линии задать численно:
      border-width:10px
    </div>
    <br>
    <div id="s3">Каждая сторона границы может
      иметь свою толщину:border-width:2px 4px 6px 8px
```

```
    </div>
    <br>
    <div id="s4">Часть границ можно не показывать
      вовсе: border-left-width:10px;
      border-bottom-width:1px
    </div>
    <br>
  </body>
</html>
```

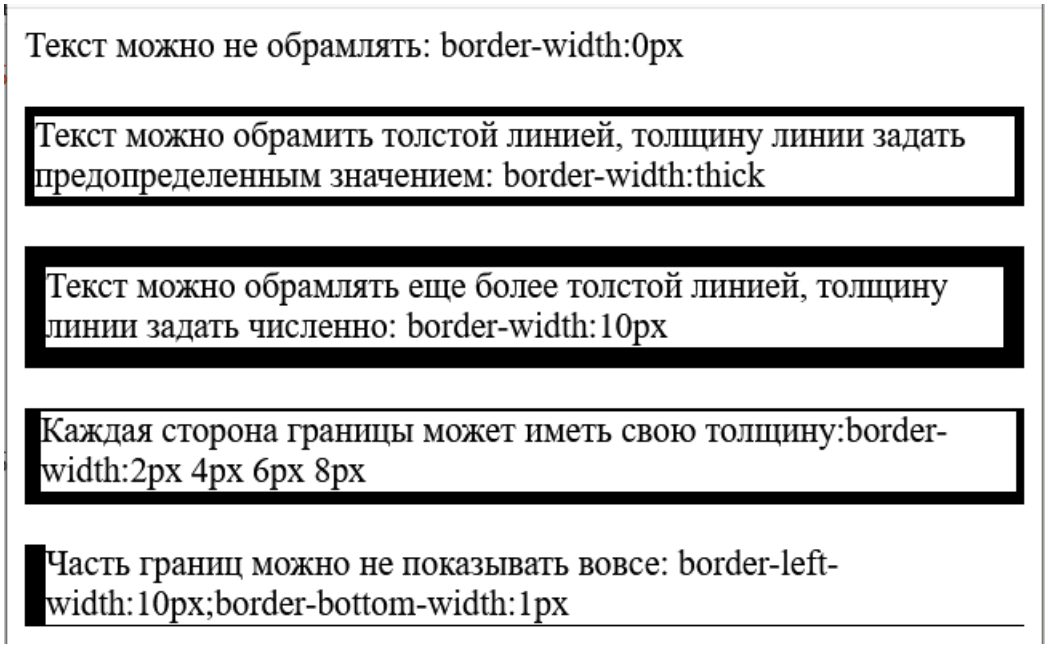


Рисунок 43.

Если для свойства border-style задано значение none, а для ширины рамки элемента установлено допустимое значение, то в этом случае рамка не выводится. Свойство не наследуется.

Рамки элемента border-color

Свойство border-color устанавливает цвет границы. Если цвет границы опущен, то он будет таким же, как и цвет текста элемента. Если в элементе нет текста, то цвет рамки будет таким же, как и цвет текста родительского элемента. Не наследуется. Параметр позволяет задать цвет границы сразу на всех сторонах элемента или только на указанных его сторонах. Разрешается использовать одно, два, три или четыре значения, разделяя их между собой пробелом. Управлять цветом одной стороны рамки можно через свойства border-top-color, border-bottom-color, border-left-color и border-right-color.

Цвет может задаваться стандартной константой или шестнадцатеричным значением RGB-палитры (рисунок 44).

```
<style>
 p {
    border-style:solid;
    border-width:8px;
   }
.s1 {
    border-width:2px;
```

```
        border-color:red;
        }
     .s2 {border-color:rgba(171,167,167,1.00) #FFFFFF;}
     .s3 {border-color: #FFFFFF #FFFFFF #FFFFFF #000000;}
    </style>
  </head>
  <body>
    <p class="s1">Важно! Текст можно выделить так</p>
    <p class="s2">Внимание! Или так.</p>
    <p class="s3">А так удобно выделять определения.</p>
  </body>
```

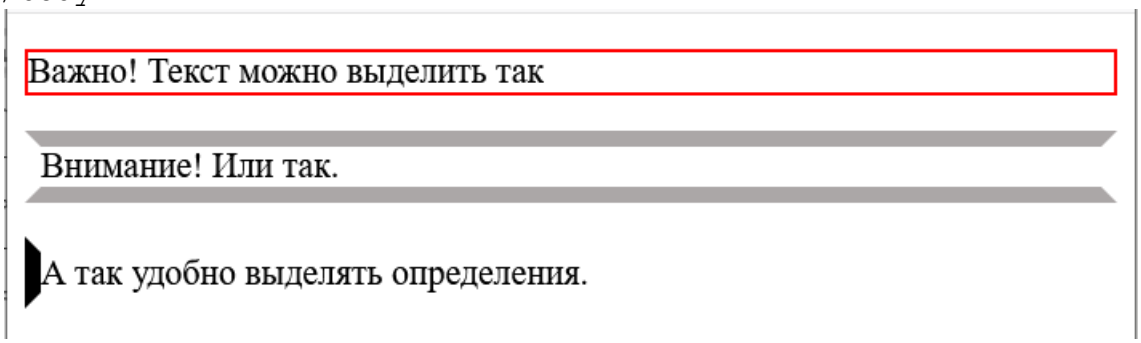


Рисунок 44.

Рамки элемента border

Свойство border можно рассматривать как сокращенный вариант одновременного использования свойств border-color, border-width и border-style. Значения этих свойств могут располагаться в любом порядке и разделяться пробелом. Не обязательно указывать значения всех трех свойств, можно указать одно, два или все три свойства. Значение по умолчанию `border:medium none`.

Для установки границы только на определенных сторонах элемента можно воспользоваться параметрами border-top, border-bottom, border-left, border-right.

Ниже приведен пример использования границ для оформления объявления.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Документ без названия</title>
    <style>
      article {
         background-color: #33A1B6;
         border: 5px solid #0b385f;
         border-bottom-style: dashed;
         }
      h2 {
         color: #3E1F97;
         padding-left: 60px;
         margin-top: 20px;
         border-top: 2px dotted #2FDECA;
         border-bottom: 1em double rgba(43,194,15,1.00);
         }
    </style>
```

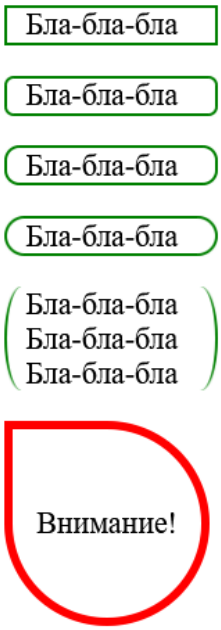
```
  </head>
  <body>
    <article>
     <h2>Внимание!</h2>
     <p>Сроки зачетов переносятся. Бла-бла-бла.</p>
    </article>
  </body>
</html>
```

Рамки элемента border-radius

Свойство border-radius используется для закругления углов элементов. Свойство применяется даже если элемент не имеет границ, в этом случае оно распространяется на фон. Можно задать значения свойства через /, в этом случае первое число определяет горизонтальный, в второе – вертикальные радиусы эллипса.

Если задать два значения для свойства border-radius, то первое значение закруглит верхний левый и нижний правый угол, а второе – верхний правый и нижний левый. Общее свойство border-radius позволяет закруглить все углы одновременно, а с помощью свойств border-top-left-radius, border-top-right-radius, border-bottom-right-radius, border-bottom-left-radius можно закруглить каждый угол отдельно. Ниже приведены примеры применения обрамления границ блоков.

```
<!doctype html>
<html>
  <head>
    <style>
      p {
         border: 2px solid green;
         width: 100px;
         padding-left: 10px;
        }
      .round1 {border-radius: 5px;}
      .round2 {border-radius: 8px;}
      .round3 {border-radius: 12px;}
      .round4 {
         border-top: none;
         border-bottom: none;
         border-radius:
         30px/90px;
         }
      .atention{
         padding: 0;
         width: 100px;
         height: 100px;
         border: 5px solid #FF0000;
         text-align: center;
         line-height: 100px;
         border-radius: 0 50% 50% 50%;
         }
    </style>
  </head>
```



```
  <body>
    <p>Бла-бла-бла</p>
    <p class="round1">Бла-бла-бла</p>
    <p class="round2">Бла-бла-бла</p>
    <p class="round3">Бла-бла-бла</p>
    <p class="round4">
     Бла-бла-бла<br>
     Бла-бла-бла<br>
     Бла-бла-бла<br>
     </p>
    <p class="atention">Внимание!</p>
  </body>
</html>
```

Закругление можно применять не только к границам, но и к самим блокам. Это удобно при оформлении пунктов меню (рисунок 45).

```
<!doctype html>
<html>
  <head>
    <style>
      .menuitem {
              width: 165px;
              float: left;
              background-color: #555555;
              color: #ffffff;
              list-style-type: none;
              margin: 4px;padding: 2px;
              text-align: center;
              cursor: pointer;
              border-radius: 15px;
              }
      .menuitem:hover {
              background-color:rgba(215,214,214,1.00) ;
              }
    </style>
  </head>
  <body>
    <menu>
      <ul >
        <li  class="menuitem">Пункт 1</li>
        <li  class="menuitem">Пункт 2</li>
        <li  class="menuitem">Пункт 3</li>
        <li  class="menuitem">Пункт 4</li>
      </ul>
    </menu>
  </body>
</html>
```

Рисунок 45.

Рамки-изображения border-image

В качестве рамки можно установить симметричное изображение. Для этого используется свойство border-image, которое в свою очередь является краткой записью свойств border-image-source, border-image-slice, border-image-width, border-image-outset и border-image-repeat.

Border-image-source задаёт путь к изображению, которое будет использоваться для оформления границ блока.

Свойство border-image-width задаёт ширину изображения для границы элемента. Если ширина не задана, то по умолчанию она равна 1.

Свойство border-image-outset позволяет сместить рамку-изображение за пределы границ элемента на указанную длину.

Размер угловой части изображения, задаётся с помощью значения свойства border-image-slice.

Свойство border-image-repeat управляет заполнением фоновым изображением пространства между углами рамки. Значение по умолчанию stretch растягивать изображение на все пространство блока.

1.3.25. Внешний контур outline

Контур – это линия, которая проводится вокруг элементов за пределами границ и повторяет форму элемента. Контуры широко используются для улучшения пользовательского интерфейса. Например, контуры позволяют выделять активные элементы интерфейса, такие как, кнопки, поля формы, карты изображений и т.п.

Контур всегда выводится поверх всех элементов и не влияет на положение или размер блока или любых других блоков. Поэтому отображение или скрытие контуров не вызывает перекомпоновку страницы.

Свойство outline представляет краткую запись свойств outline-color, outline-style и outline-width, которые имеют такой же смысл, как и аналогичные свойства для рамки border-color, border-style и border-width.

Значение по умолчанию `outline: invert none medium`. Значение invert выполнит инверсию цвета пикселей на экране. Это обеспечивает видимость границы фокуса независимо от цвета фона.

На рисунке 46 при помощи свойства outline выделяются ячейки таблицы над которыми находится курсор.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
      table, th, td {
        border: 1px solid black;
        border-collapse: collapse;}
      td:hover{outline: 2px solid red;}
    </style>
  </head>
  <body>
```

```
    <table>
      <tr>
        <th> фио </th>
        <th> алг. </th>
        <th> геом. </th>
        <th> мат.ан. </th>
        <th> прогр. </th>
      </tr>

      <tr>
        <td> Иванов И. И.</td>
        <td> 8 </td>
        <td> 7 </td>
        <td> 9 </td>
        <td> 8 </td>
      </tr>

      <tr>
        <td> Петров П. П.</td>
        <td> 5 </td>
        <td> 9 </td>
        <td> 6 </td>
        <td> 5 </td>
      </tr>

    <tr>
        <td> Сидоров С. С.</td>
        <td> не явился </td>
        <td> 4 </td>
        <td> 4 </td>
        <td> 7 </td>
      </tr>
    </table>
  </body>
</html>
```

фио	алг.	геом.	мат.ан.	прогр.
Иванов И. И.	8	7	9	8
Петров П. П.	5	9	6	5
Сидоров С. С.	не явился	4	4	7

Рисунок 46.

1.3.26. Обтекание. Свойство float

Свойство float устанавливает горизонтальное выравнивание элемента относительно родительского элемента, при этом остальные элементы должны обтекать его с других сторон.

Когда-то это был основной способ стилизации html страницы.

Свойство float может принимать следующие значения:

- left – выравнивание элемента по левому краю, а все остальные элементы, например, текст, огибают его по правой стороне;
- right – выравнивание элемента по правому краю, а все остальные элементы, например, текст, огибают его по левой стороне;
- none – отсутствие выравнивания текущего элемента, элемент выводится там, где он задан. Является значением по умолчанию;
- inherit – наследует значение свойства от родительского элемента.

Управляя обтеканием можно смещать «плавающие блоки» влево или вправо на текущей строке. Плавающий блок смещается влево или вправо до тех пор, пока его внешний край не коснется края содержащего блока или внешнего края другого плавающего блока. Если для плавающего блока недостаточно места по горизонтали, он будет смещаться вниз до тех пор, пока не уместится.

В приведенном ниже примере (рисунок 47) сначала выводится блок с атрибутом class="normal1". Он находится в нормальном потоке, имеет заданную высоту и зеленую границу. Это отдельно стоящий блок. Он занимает всю доступную ему ширину родительского элемента. Затем выводится плавающий блок с атрибутом class="floating_block". Задано выравнивание по левому краю. Заданы высота и ширина блока и красная граница. Следующий, третий блок с атрибутом class="normal2". Он также находится в нормальном потоке, и имеет синюю границу. Блок занимает всю доступную ему ширину родительского элемента, он игнорирует плавающий блок, а текст в третьем блоке обтекает плавающий блок справа.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Свойство float</title>
    <style>
      .normal1{
          height: 100px;
          border: 2px solid #75EFC3;
          }
      .floating_block{
          height: 200px;
          width: 200px;
          border: 2px solid #F5171A;
          float: left;
          }
      .normal2{
          height: 100px;
          border: 2px solid #564DE9;
          }
    </style>
  </head>
  <body>
    <div class="normal1">Это отдельно стоящий блок.
      Он находится в нормальном потоке. Занимает всю
      доступную ему ширину родительского элемента.
```

```
    </div>
    <div class="floating_block">Это плавающий блок. У
      него задана определенная ширина и выравнивание по
      левому краю.
    </div>
    <div class="normal2">Это отдельно стоящий блок. Он
      находится в нормальном потоке.
    </div>
  </body>
</html>
```

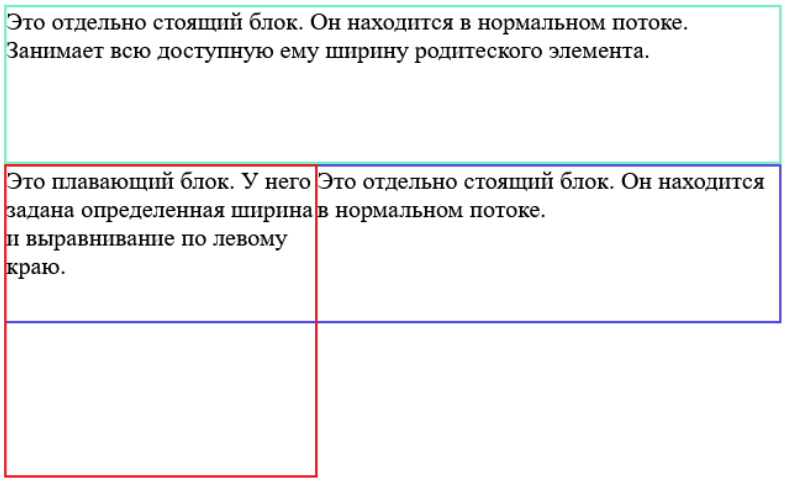


Рисунок 47.

Если изменить высоту блоков и добавить в третий блок больше текста, то обтекание будет более очевидно (рисунок 48).

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Свойство float</title>
    <style>
      .normal1{
          height: 100px;
          border: 2px solid #75EFC3;}
      .floating_block{
          height: 80px;
          width: 200px;
          border: 2px solid #F5171A;
          float: left;}
      .normal2{
          height: 200px;
          border: 2px solid #564DE9;}
    </style>
  </head>
  <body>
    <div class="normal1">Это отдельно стоящий блок. Он
      находится в нормальном потоке. Занимает всю
      доступную ему ширину родительского элемента.
    </div>
    <div class="floating_block">Это плавающий блок.
      У него задана определенная ширина и выравнивание по
```

```
      левому краю.
    </div>
    <div class="normal2">Это отдельно стоящий блок. Он
      находится в нормальном потоке. Это отдельно стоящий
      блок. Он находится в нормальном потоке. Это отдельно
      стоящий блок. Он находится в нормальном потоке. Это
      отдельно стоящий блок. Он находится в нормальном
      потоке.
    </div>
  </body>
</html>
```

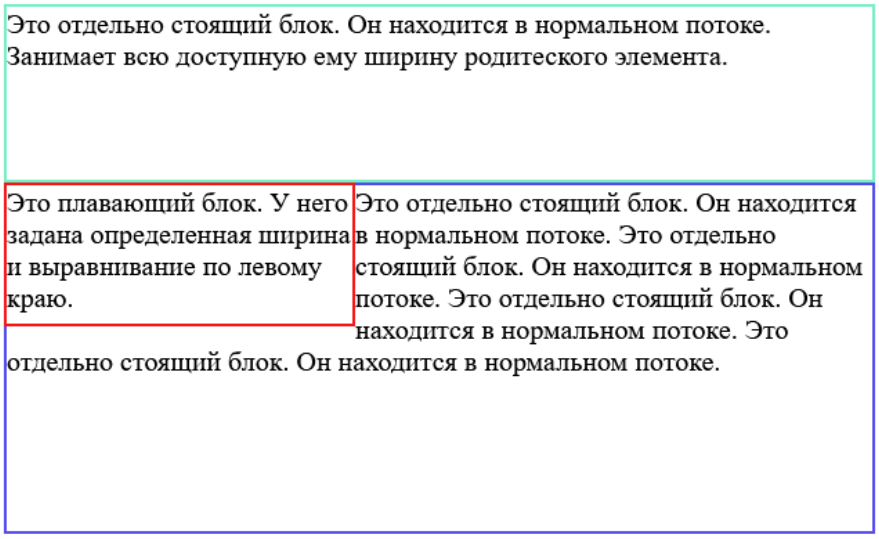


Рисунок 48.

1.3.27. Обтекание. clear

Свойство clear запрещает обтекание элемента другими элементами разметки относительно родительского элемента.

Свойство clear может принимать следующие значения:

- both – запрет на обтекание слева и справа;
- left – запрет на обтекание слева;
- right – запрет на обтекание справа;
- none – отменяет запрет на обтекание, установленное значениями both, left, right, обтекание возобновляется согласно старым установкам. Если отменить обтекание, автоматически происходит переход на следующую строку.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
      span {border:2px solid;}
      span.s1 {float:left;}
      span.s2 {float:right;}
    </style>
  </head>
  <body>
    <div>
```

```
      <span class="s1">Блок 1</span>
      <span class="s1" style="clear: left;">Блок 2</span>
      <span class="s1" style="clear: right;">Блок 3</span>
      <span class="s1" style="clear: both;">Блок 4</span>
      <span class="s1">Блок 5</span>
      <span class="s2">Блок 6</span>
      <span class="s2" style="clear: left;">Блок 7</span>
      <span class="s2" style="clear: right;">Блок 8</span>
      <span class="s2" style="clear: both;">Блок 9</span>
      <span class="s2">Блок 10</span>
    </div>
  </body>
</html>
```

В приведенном примере (рисунок 49) определено два стиля для тега , в первом значение свойства float установлено в left, а во втором в right. Для вывода блоков с первого по пятое используется первый стиль, а для вывода последующих пяти блоков – второй стиль. При выводе первого блока свойство clear не используется, блок ведет себя как встроенный и располагается в начале абзаца. Для вывода второго блока свойство clear установлено в left, поэтому он выводится таким образом, чтобы его верхний край находился ниже нижнего внешнего края всех ранее созданных левосторонних перемещаемых блоков. Фактически зрительно это привело к переходу на следующую строку. Для третьего блока свойство clear установлено в right, что означает, что этот блок должен быть выведен ниже любого нижнего внешнего края любого правостороннего перемещаемого блока, который был сгенерирован в исходном документе предыдущим элементом. Правосторонних перемещаемых блоков еще не выводилось, поэтому третий блок становится справа за вторым во второй линии. При выводе четвертого слова установлен запрет на обтекание слева и справа. Поэтому он помещается ниже блока 2 (clear:left) и ниже блока 3 (clear:right), а именно в третий ряд. Для пятого блока свойство clear не используется, блок ведет себя как встроенный и становится на одну линию с предыдущим, четвертым блоком, блоком сразу за ним.

Для вывода слов с шестого по десятое используется второй стиль, где значение свойства float установлено в right. При выводе шестого слова свойство clear не используется, блок ведет себя как встроенный и располагается в самой правой позиции в той же линии, где уже есть четвертый и пятый блоки. Для вывода седьмого блока свойство clear установлено в left, поэтому он выводится таким образом, чтобы его верхний край находился ниже нижнего внешнего края всех ранее созданных левосторонних перемещаемых блоков, а именно блока 4, поскольку у него обтекание запрещено с обоих сторон. Поэтому седьмой блок становится самым правым на четвертой линии. Для восьмого блока свойство clear установлено в right, поэтому он быть выведен ниже любого нижнего внешнего края любого правостороннего перемещаемого блока, который был сгенерирован в исходном документе предыдущим элементом, т.е. ниже блока семь на пятой линии. По этой же причине блок 9 выводится самым крайним справа на шестой линии, поскольку у него clear установлено в both. Для десятого блока свойство clear не используется, блок ведет себя как встроенный и

становится на одну линию с предыдущим, девятым блоком, сразу слева перед ним. Результат выполнения представлен ниже.

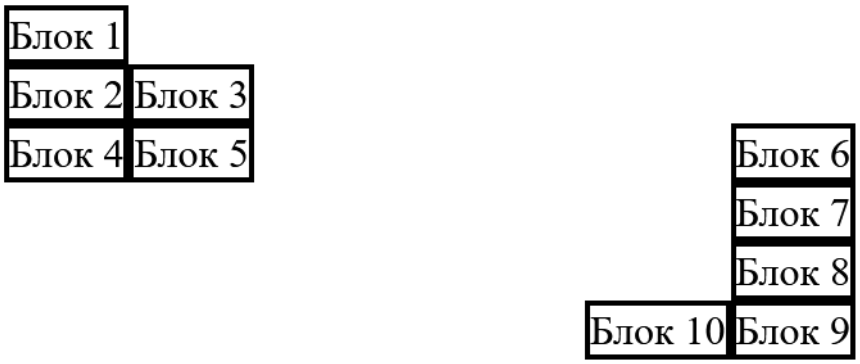


Рисунок 49.

1.3.28. Flexbox (Flexible Box Layout) в CSS

Flexbox — это современный способ выстраивания элементов в контейнере, особенно полезный для адаптивной и динамической верстки. Он особенно полезен при разработке интерфейсов, которые должны корректно отображаться на разных устройствах и экранах.

Flexbox заменяет устаревшие методы с float и inline-block, упрощает адаптивную верстку, идеален для программистов при создании навигационных панелей, карточек, галерей и форм.

Чтобы использовать Flexbox, достаточно задать любому блоку display: flex: Теперь все дочерние элементы становятся flex-элементами и подчиняются правилам Flexbox.

Есть несколько онлайн-игр чтобы потренироваться использовать flex для размещения блоков: игра1: https://flexboxfroggy.com/#ru, игра2: http://cssgridgarden.com/#ru

Основные свойства контейнера

На рисунке 50 указаны как основные свойства контейнера влияют на размещение элементов по горизонтали и вертикали.

flex-direction: определяет направление главной оси (row, column, row-reverse, column-reverse)

justify-content: выравнивание по главной оси (flex-start, center, space-between, space-around, space-evenly)

align-items: выравнивание по поперечной оси (stretch, center, flex-start, flex-end) на заданной строке. Это своеобразная версия ***justify-content***, но только для поперечной оси, которая перпендикулярна главной

align-content: выравнивает и распределяет строки контейнера. свойство не приности эффекта, когда есть только одна строка flex элементов.

flex-wrap: разрешает перенос строк (nowrap, wrap, wrap-reverse)

gap: задаёт расстояние между элементами

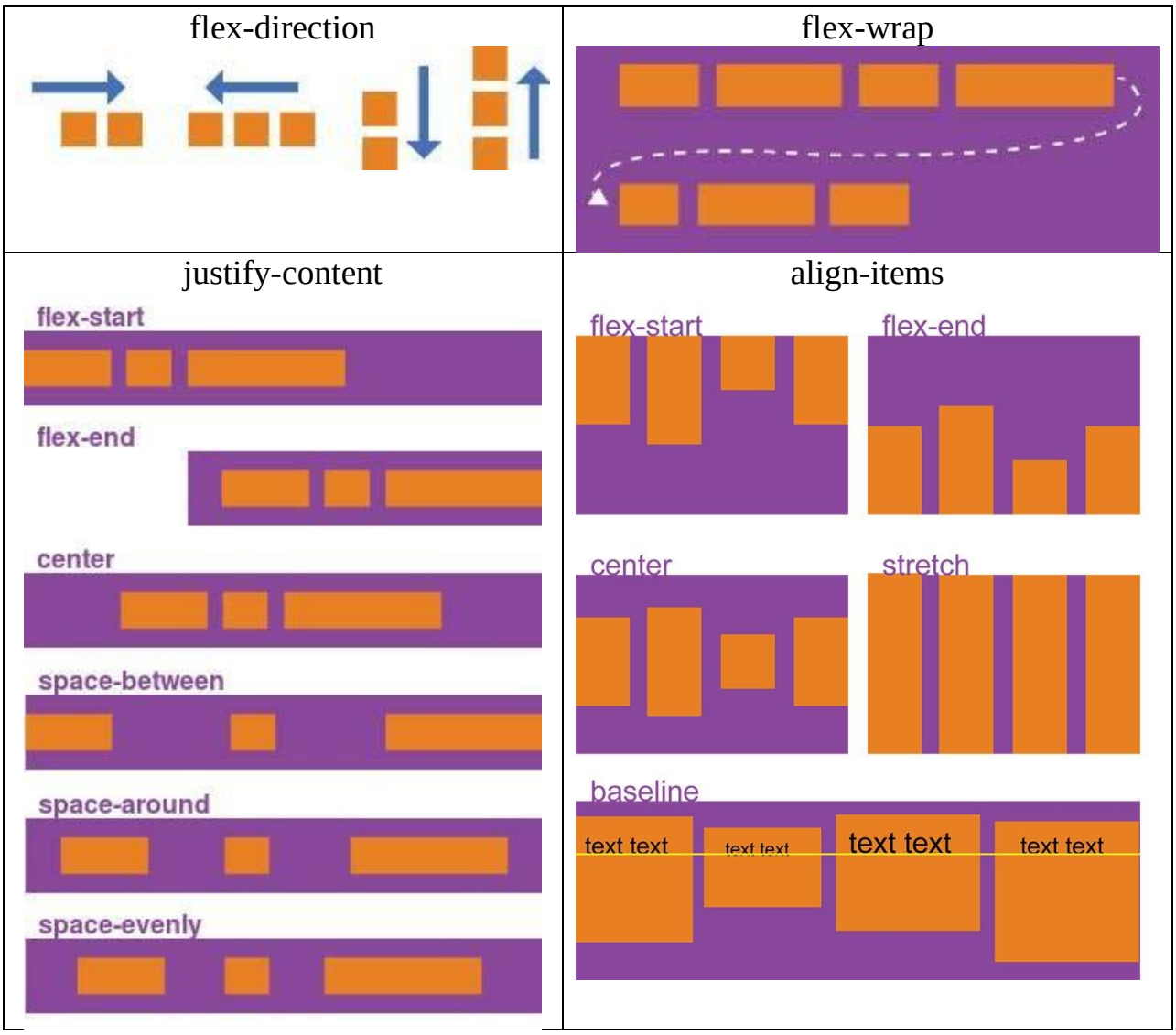


Рисунок 50.

Свойства flex-элементов

Каждый элемент внутри контейнера может управлять своим размером и поведением:

flex: сокращённая запись: flex: grow shrink basis
flex-grow: определяет, насколько элемент может увеличиваться
flex-shrink: определяет, насколько элемент может уменьшаться
flex-basis: начальный размер элемента до распределения пространства
align-self: индивидуальное выравнивание элемента по поперечной оси

Пример использования.

начнем с простого примера, решающего ежедневную проблему — идеальное центрирование.

```
.parent {
  display: flex;
  height: 300px; /* Неважно */
}
```

```
.child {
  width: 100px;  /* Неважно */
  height: 100px; /* Неважно */
  margin: auto;  /* Магия! */
}
```

Суть в том, что margin с параметром auto в flex контейнере поглощает свободное пространство. Таким образом, выставляя вертикальный отступ на auto, вы заставите элементы идеально сцентрироваться по двум осям.

Пример. Меню в котором адаптивность достигается **автоматически:**

display: flex + flex-wrap: wrap позволяет элементам переноситься на новую строку, если экран узкий.

flex: 1 1 auto делает ссылки гибкими: они растягиваются равномерно, но могут сжиматься.

```
<nav class="menu">
  <a href="#">Главная</a>
  <a href="#">О нас</a>
  <a href="#">Услуги</a>
  <a href="#">Контакты</a>
</nav>

.menu {
  display: flex;
  flex-wrap: wrap;
  gap: 10px;
  padding: 10px;
  background: #333;
}

.menu a {
  flex: 1 1 auto;
  text-align: center;
  padding: 10px;
  color: white;
  text-decoration: none;
  background: #555;
  border-radius: 5px;
  transition: background 0.3s, transform 0.2s;/* эффект плавный*/
}

.menu a:hover {
  background: #1abc9c;/* меняет фон при наведении*/
  transform: scale(1.05);/* слегка увеличивает элемент*/
}
```

Пример создания “таблицы” N на N для разработки игры “15”. Считаем, что мы сгенерировали html разметку следующего вида:

```
<div class="grid">
  <div>1</div>
  <div>2</div>
```

```
    <div>3</div>
  ..............
    <div>24</div>
    <div>N*N</div>
  </div>

  7:root {
    --N: 4; /* 3, 5, 6 — всё будет работать */
  }

  .grid {
    display: flex;
    flex-wrap: wrap;
    gap: 4px;
    max-width: 400px;
    margin: auto; /*простой способ центрировать блок с фиксированной
шириной*/
    padding: 4px;
    background: #333;
  }

  .grid > div {
    flex: 0 0 calc(100% / var(--N) - 4px);
    aspect-ratio: 1 / 1;
    background: #eee;
    display: flex;
    justify-content: center;
    align-items: center;
    font-size: 2em;
    cursor: pointer;
    user-select: none;
    transition: background 0.2s;
  }

  .grid > div:hover {
    background: #ddd;
  }
```

1.3.29. CSS цвета

Цвет в CSS может определяться как:

- константы;
- значения RGB или RGBA;
- шестнадцатеричные значения;
- значения HSL и HSLA (CSS3);
- ключевым словом currentcolor.

При выборе цветовой схемы для своего веб-сайта важно убедиться, что контрастность между цветом фона и цветом текста, размещенного над ним, достаточно высока, чтобы люди, испытывающие проблемы со слабым зрением, могли прочитать содержимое страницы. Дизайн сайта может выглядеть отлич-

но, но быть не слишком удобным при чтении текста. Или люди с нарушениями зрения, такими как дальтонизм, не смогут читать контент.

Существует простой способ проверить, достаточно ли хороший контраст, чтобы не вызывать проблем с восприятием. В Интернете существует множество инструментов для проверки контрастности, например, WebAIM (рисунок 51) https://webaim.org/resources/contrastchecker/.

Кроме того, высокий коэффициент контрастности также позволяет пользователям с устройствами, имеющими глянцевый экран, например, смартфон или планшет, а также лучше читать страницы в условиях яркого освещения, такого как солнечный свет.

Коэффициент контрастности цвета определяется путем сравнения значений яркости текста и цвета фона. Для текстового содержимого требуется соотношение 4,5 : 1 и 3 : 1 для более крупного текста, такого как заголовки. А крупный текст определяется как 18.66px жирным шрифтом или больше, или 24px или больше. Пример сравнения контрастности на https://webaim.org/resources/contrastchecker/.

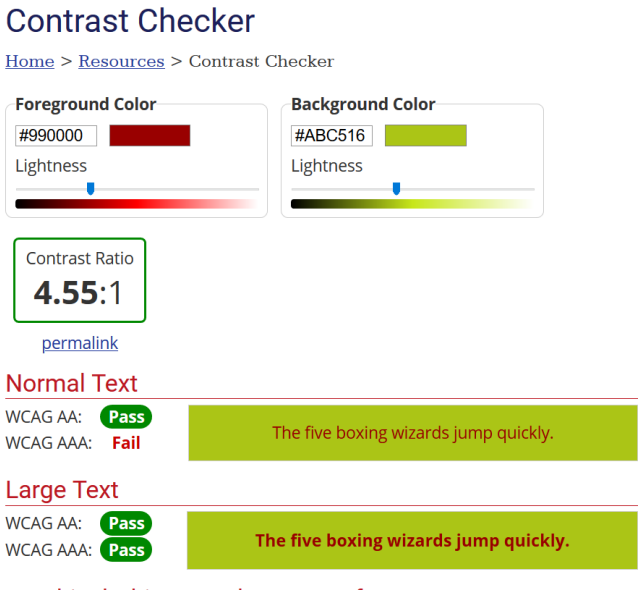


Рисунок 51 – Сравнение контрастности.

1.3.30. color

Свойство color устанавливает цвет символов текста элемента разметки. Цвет может задаваться стандартной константой или шестнадцатеричным значением RGB-палитры.

Константы

Цвет может быть задан через имя константы, например, "red". HTML и CSS поддерживает 140 стандартных названия цветов (http://www.w3schools.com/colors/colors_names.asp). Названия цветов нечувствительны к регистру.

Несмотря на то, что названия цветов могут быть заданы 140 константами, HTML распознает только 16 основных цветовых ключевых слов (рисунок 52), определенных еще в CSS1, остальные значения обрабатываются используя спе-

циальный алгоритм для преобразования нераспознанных значений, иногда это совершенно разные цвета в различных браузерах. В CSS могут использоваться и все остальные ключевые слова для обозначения цвета. Если в HTML, неизвестные ключевые слова для обозначения цвета обрабатываются используя специальный алгоритм, то в CSS они будут полностью проигнорированы. Все ключевые слова цвета представляют собой сплошные цвета без прозрачности.

Несколько ключевых слов являются псевдонимами друг для друга. Например, aqua и cyan, fuchsia и magenta и т. д.

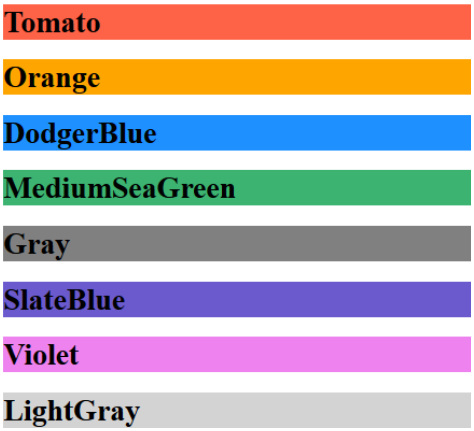


Рисунок 52.

RGB цвета

В CSS цвет можно указать как значение RGB, используя следующую формулу: RGB (красный, зеленый , синий). Каждый параметр (красный, зеленый и синий) определяет интенсивность цвета от 0 до 255.

Оттенки серого определяются с использованием равных значений для всех 3 источников света (рисунок 53):

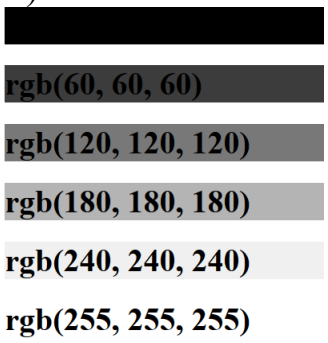


Рисунок 53.

RGBA цвета

Можно задать цвет как значение RGBA, используя следующую формулу:rgba(красный, зеленый , синий, прозрачность).

Параметры красный, зеленый и синий определяют интенсивность цвета от 0 до 255, Могут выражаться процентами, число 255 соответствует 100% . Прозрачность может быть числом от 0 до 1 или процентами, где число 1 соответствует 100% (полная непрозрачность).

```
rgba (51, 170, 51, .1) / * 10% непрозрачный зеленый * /
rgba (51, 170, 51, .4) / * 40% непрозрачный зеленый * /
RGBA (51, 170, 51, 0,7) / * 70% непрозрачный зеленый * /
rgba (51, 170, 51, 1) / * полностью непрозрачный зеленый * /
```

Рисунок 54. Пример прозрачности

Шестнадцатеричные значения

В CSS цвет можно указать с помощью шестнадцатеричного значения в форме: # rrggbb. (рисунок 55) Где rr (красный), gg (зеленый) и bb (синий) – шестнадцатеричные значения от 00 до ff. Например, # ff0000 задаёт красный цвет, потому что для красного установлено самое высокое значение (ff), а для других – самое низкое (00).

Рисунок 55.

HSL и HSLA цвета

Цветовая модель HSL определяет цвет в соответствии с его компонентами оттенка, насыщенности и яркости. Необязательный альфа-компонент представляет прозрачность цвета.

```
hsl(оттенок, насыщенность, яркость).
hsla(оттенок, насыщенность, яркость, прозрачность).
```

Оттенок представляет собой угол (задаётся в градусах без указания единицы измерения), в диапазоне от 0° или 360° цветового круга относительно красного цвета. Красный цвет имеет угол 0° или 360°. Зеленый – 120°. Синий – 240°. Отрицательные значения допускаются, например 300° можно записать как -60°.

Насыщенность (saturation) и яркость (lightness) задаются в процентах.

Цвета могут быть яркими (насыщенными) или тусклыми. Чем меньше цвета, тем более серый оттенок получается. 0% насыщенности цвета – это серый, 100% – чистый цвет.

Яркость (lightness) добавляет к цвету белый или черный, чтобы изменить его. Значение параметра яркости меньше 50% означает, что добавляется черный цвет, выше – белый. Тон при этом остается неизменным.

Прозрачность может быть задана числом от 0 до 1 или процентами, где число 1 соответствует 100% (полная непрозрачность).

Многие дизайнеры считают HSL более интуитивным, чем RGB, поскольку он позволяет настраивать оттенок, насыщенность и яркость каждого цвета независимо. HSL также может упростить создание набора подходящих цветов (например, когда вам нужно несколько цветов одного оттенка). Цветовой круг представлен на рисунке 56.

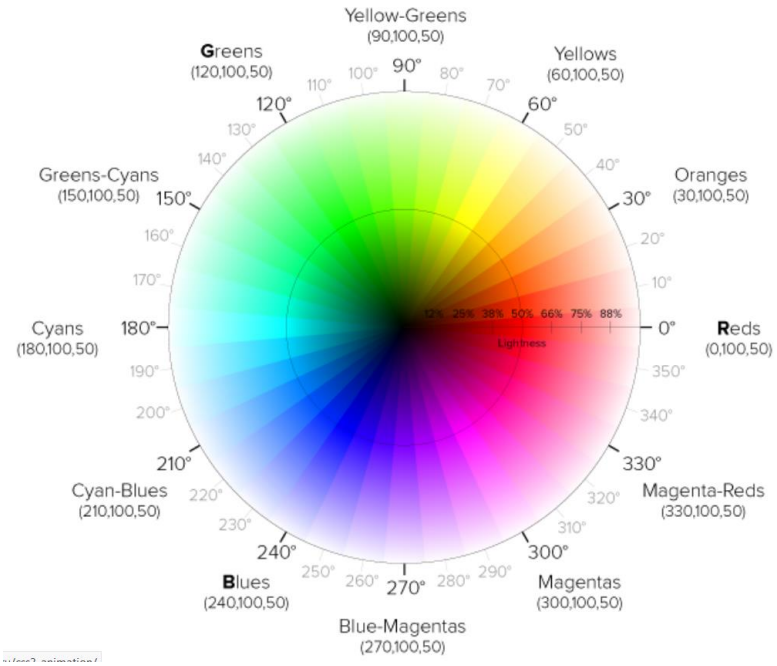


Рисунок 56 – цветовой круг.

Чтобы получить (рисунок 57) черный цвет, нужно присвоить показателям оттенка, насыщенности и яркости нулевое значение – hsla(0, 0%, 0%, 1). Белый цвет получается при 100%-ном значении яркости hsla(0, 0%, 100%, 1), а серый цвет – при нулевом значении насыщенности hsla(0, 0%, 50%, 1).

0° красный					
Насыщенность					
0%	25%	50%	75%	100%	Яркость
					100%
					88%
					75%
					63%
					50%
					38%
					25%
					13%
					0%

Рисунок 57.

Есть конвертеры цветов из одного представления в другое. Например, https://colorscheme.ru/color-converter.html.

Кроме этого данный ресурс позволяет выбрать цвет, а к нему автоматически подбирается один или несколько (по выбору) подходящих цветов и показывается, как выбранные цвета и их варианты сочетаются между собой.

currentcolor

Если у блока указать цвет границы как ***currentcolor***, то граница будет такого цвета, как цвет текста в блоке.

```
body {color: grey;}
div {border: 1px solid currentcolor;}
```

1.3.31. Фон элементов

У каждого html-элемента есть слой фона. Фон может быть полностью прозрачным, это значение по умолчанию, а может быть залит цветом и/или одним или несколькими изображениями.

Свойства фона не наследуются. Фон не отображается у пустых элементов с нулевой высотой. Отрицательные значения свойства margin не влияют на фон элемента. Фоном элементов управляет общее свойство background, которое является сокращенным свойством для: background-color, background-image, background-position, background-size, background-repeat, background-origin, background-clip, background-attachment.

background-color

Свойство background-color устанавливает цвет фона элемента разметки. Если есть фоновый рисунок, то цвет заливается за фоновым изображением. Для блочных элементов цвет фона распространяется на всю ширину и высоту блока элемента, включая padding и border, но не margin, для строчных – только на область их содержимого.

Свойства фона не наследуются, но если никаких свойств фона элемента не менять, то фон родительского блока будет просвечивать (рисунок 58), поскольку по умолчанию значение background-color установлено в transparent.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
      h1 {
          background-color: BlueViolet;
          color:DarkTurquoise;
         }
     div {background-color: DarkKhaki;}
     p {background-color: yellow;}
    </style>
  </head>
  <body>
    <h1>Пример CSS background-color.</h1>
    <div>
      Это текст внутри тега div.
      <p>Это параграф. У него есть собственный фон.</p>
      Это еще тег div.
    </div>
  </body>
</html>
```

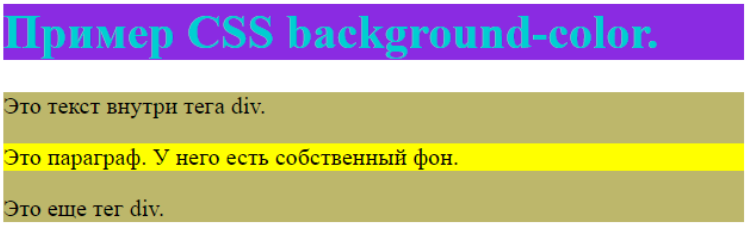


Рисунок 58.

background-image

Свойство background-image определяет одно или несколько фоновых изображение для элемента. Если одновременно для элемента задан и цвет фона, и фоновое изображение, то фон заданного цвета будет виден, пока фоновая картинка не загрузится полностью. Даже если изображения непрозрачны и цвет не будет отображаться в нормальных условиях, веб-разработчики всегда должны указывать background-color. Если изображения не могут быть загружены, например, когда сеть не работает, цвет фона будет использоваться в качестве запасного варианта. Значением свойства является адрес файла изображения. Размер фонового рисунка не подстраивается автоматически под размер элемента разметки. Если размер изображения меньше размера элемента разметки, то по умолчанию фоновый рисунок повторяется (рисунок 59). Если размер изображения больше размера элемента разметки, то будет видна только часть фонового рисунка. При использовании фонового изображения надо использовать изображение, которое не мешает тексту.

```
<!doctype html>
<html>
 <head>
   <style>
    body {   background-image: url(images/bgdesert.jpg);}
    .paper {    background-image: url(images/paper.gif);}
   </style>
 </head>
 <body>
  <div >Не удачный фон. Текст практически не читаем.</div>
  <div class="paper">С таким фоном читабельность хорошая.</div>
 </body>
</html>
```

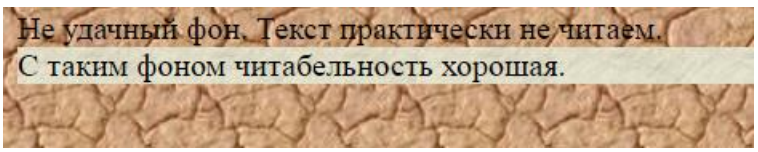


Рисунок 59.

background-repeat

Свойство background-repeat определяет, будет ли дублироваться фоновое изображение, установленное с помощью параметра background-image, если размер изображения меньше размера элемента разметки.

Свойство background-repeat может принимать следующие значения:

repeat – изображение дублируется по вертикали и горизонтали так часто, чтобы покрыть область отрисовки фона. Если изображение не помещается, оно обрезается. Является значением по умолчанию.

repeat-x – изображение дублируется только по горизонтали.

repeat-y – изображение дублируется только по вертикали.

no-repeat – изображение не дублируется.

space – изображение дублируется столько раз, сколько оно помещается в области фона, не обрезаясь, изображения расположены на равном расстоянии друг от друга. Первое и последнее изображения касаются краев области. Если область рисования фона больше, чем область позиционирования фона, шаблон повторяется, чтобы заполнить область рисования фона. Если недостаточно места для двух копий изображения, то размещается только одно изображение, а свойство background-position определяет его положение.

- round – изображение повторяется так часто, чтобы заполнить область фона, масштабируясь и не обрезаясь.
- no-repeat – изображение размещается один раз и не повторяется.
- initial – устанавливает значение свойства в значение по умолчанию.
- inherit – наследует значение свойства от родительского элемента.

Не всегда значение по умолчанию уместно. Повторяемые градиентные изображения для фона смотрятся странно.

background-position

Свойство background-position содержит координаты левого верхнего угла фонового изображения. Это комбинированное свойство, заменяющие атрибуты background-position-x и background-position-y. Значения могут задаваться в процентах и абсолютных координатах. Если задана только одна координата, то она считается горизонтальной, а для вертикальной координаты принимается значение 50%. Значение по умолчанию 0% 0%. Ниже приведены результаты использования свойства.

background-attachment

Свойство background-attachment определяет, будет ли прокручиваться фоновое изображение вместе с содержимым элемента или будет зафиксировано. Применение этого свойства имеет смысл, если контент элемента достаточно большой и целиком не виден, а для полного просмотра содержимого есть полосы прокрутки.

Допустимы следующие значения свойства:

- scroll – фоновое изображение прокручивается вместе с контентом;
- fixed – фоновое изображение остается неподвижным во время прокрутки содержимого элемента.
- local – фоновое изображение прокручивается вместе с содержимым элемента;
- initial –устанавливает значение свойства в значение по умолчанию;
- inherit –наследует значение свойства от родительского элемента.

background-clip

Свойство background-clip определяет область рисования фона. Фон всегда рисуется под рамкой элемента, если таковая имеется.

Корневой элемент <html> имеет другую область рисования фона, поэтому свойство background-clip на него не влияет.

Возможны следующие значения свойства:

border-box – фон закрашивает область в пределах рамки элемента, значение по умолчанию;

padding-box – фон закрашивает область в пределах внутренних полей элемента;

content-box – фон закрашивает только область содержимого;

initial – устанавливает значение свойства в значение по умолчанию;.

inherit – наследует значение свойства от родительского элемента.

background-origin

Определяет исходную позицию (область позиционирования фона) фонового изображения. Допустимые значения (рисунок 60):

padding-box – фон позиционируется относительно верхних границ области внутренних полей элемента. Значение по умолчанию.

border-box – фон позиционируется относительно верхних границ рамки элемента.

content-box – фон позиционируется относительно верхних границ области содержимого элемента.

initial – устанавливает значение свойства в значение по умолчанию.

inherit – наследует значение свойства от родительского элемента.

Рисунок 60.

свойство background-size

Свойство background-size устанавливает размер фоновых изображений. Значения свойства:

auto – значение по умолчанию, высота и ширина изображения равны его оригинальным размерам;

размер – задаётся парой значений, первое значение устанавливает ширину изображения, второе – высоту (для того, чтобы фон масштабировался вместе с текстом, размеры изображения нужно задавать в em);

% – задаёт размер фонового изображения в процентах от ширины или высоты элемента, которое заполняется фоном;.

cover – масштабирует изображение с сохранением пропорций так, чтобы его ширина или высота равнялась ширине или высоте блока;

contain – масштабирует изображение с сохранением пропорций таким образом, чтобы оно целиком поместилось внутри блока;

initial – устанавливает значение свойства в значение по умолчанию;

inherit – наследует значение свойства от родительского элемента.

background

Свойство background является объединением свойств background-color, background-image, background-position, background-size, background-repeat, background-origin, background-clip и background-attachment. Значения должны быть перечислены через пробел. Значения свойства background-size нужно записывать через слеш /, чтобы отделить его от свойства background-position. Необязательно указывать все перечисленные свойства. Сначала свойство background устанавливает всем перечисленным свойствам фона их значения, а не упомянутым атрибутам устанавливает значения по умолчанию.

1.3.32. overflow

Свойство overflow управляет отображением содержания блочного элемента, если контент элемента целиком не помещается в отведенные ему размеры. Такая ситуация может возникнуть, если для блока были установлены фиксированные значения width и height, и они оказались недостаточными.

Свойство overflow может принимать следующие значения:

- visible – полосы прокрутки не отображаются. Текст выводится полностью, а значения width и height остаются прежними, поэтому контент блока может наползать на содержимое следующего блока;
- scrool – значения width и height остаются постоянными, независимо от величины содержимого блока. При этом всегда отображаются полосы прокрутки, даже если содержимое элемента разметки полностью помещается в отведенных ей пределах;
- hidden – элемент разметки не меняет размеров, а отображается в соответствии с фиксированными значениями width и height. При этом если содержимое элемента не помещается в отведенную ему область, то оно усекается, полосы прокрутки не отображаются.
- auto – элемент разметки размеров не меняет и отображается в соответствии с фиксированными значениями width и height. При необходимости появляются полосы прокрутки.

```
<!doctype html>
<html>
  <head>
    <style>
      div{
```

```
        width:100px;
        height:100px;
        border:solid thin green;
        }
    </style>
  </head>
  <body>
    overflow:visible. Полосы прокрутки не отображаются.
    Текст отображается полностью, при этом, при
    необходимости размеры элемента увеличиваются.
    (В Mozilla отличия!!!)
    <div style="overflow:visible">Всего 10 слов.
      Слово №1. Слово №2. Слово №3. Слово №4. Слово №5.
      Слово №6. Слово №7. Слово №8. Слово №9.
      Слово №10.
    </div>
    <br>
    <br>
    overflow:hidden. Полосы прокрутки не отображаются.
    Текст усекается до размеров элемента.
    <div style="overflow:hidden">
      Всего 10 слов. Слово №1. Слово №2. Слово №3.
      Слово №4. Слово №5. Слово №6. Слово №7. Слово №8.
      Слово №9. Слово №10.
    </div>
    <br>
    <br>
    overflow:scroll. Полосы прокрутки отображаются
    всегда.
    <div style="overflow:scroll">
      Всего 3 слова. Слово №1. Слово №2.
      Слово №3.
    </div>
    <br>
    <br>
    overflow:auto. Полосы прокрутки появляются при
    необходимости.
    <div style="overflow:auto">
      Всего 10 слов. Слово №1. Слово №2. Слово №3.
      Слово №4. Слово №5. Слово №6. Слово №7.
      Слово №8. Слово №9. Слово №10.
    </div>
    <br>
    <br>
    overflow:auto. Полосы прокрутки появляются
    при необходимости.
    <div style="overflow:auto">
      Всего 3 слова. Слово №1. Слово №2. Слово №3.
    </div>
  </body>
</html>
```

В приведенном примере (рисунки 61-64) для всех блоков установлены фиксированные значения, таким образом, что содержимое блоков не помещает-

ся в отведенные им пределы. Размер блока не увеличился, но его контент выводится полностью, что приводит к наложению текста на последующие блоки и потере читабельности.

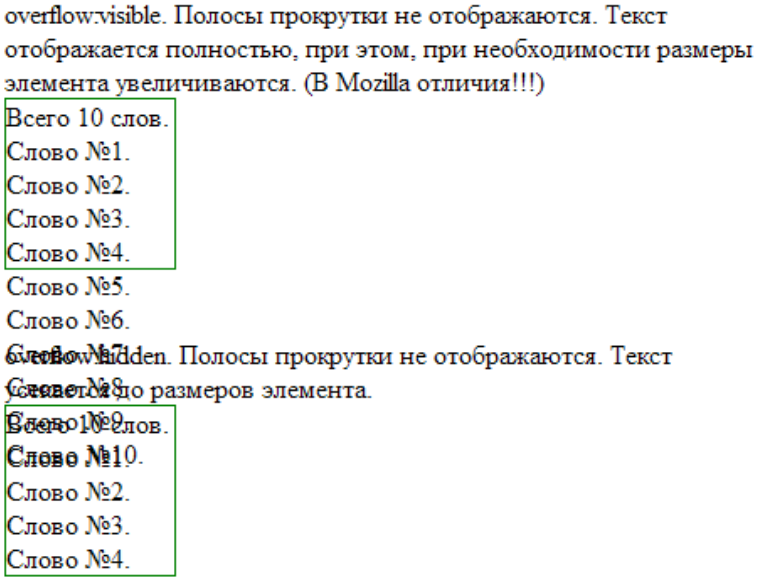


Рисунок 61.

Для второго блока `overflow:hidden`, поэтому блок не изменяет своих размеров, а его содержимое видно только частично.

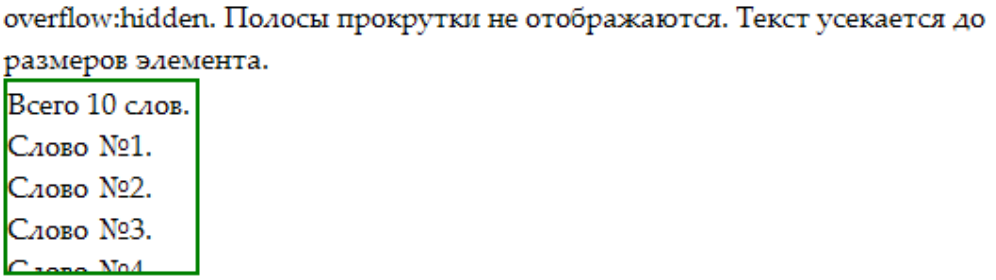


Рисунок 62.

`overflow:scroll`. В этом случае полосы прокрутки отображаются всегда, даже если в этом нет необходимости. В данном случае вертикальная полоса прокрутки необходима, а горизонтальная – нет.

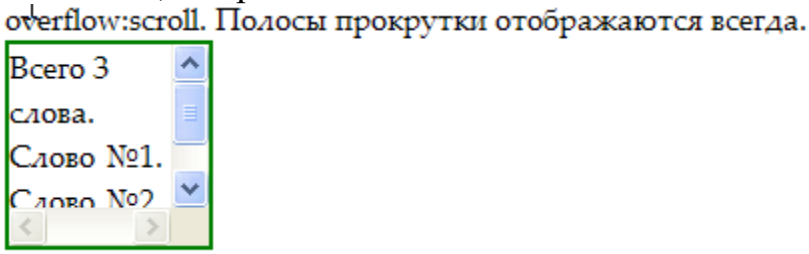


Рисунок 63.

`overflow:auto`. только вертикальная полоса прокрутки, горизонтальной полосы прокрутки нет, т. к. в ней нет необходимости.

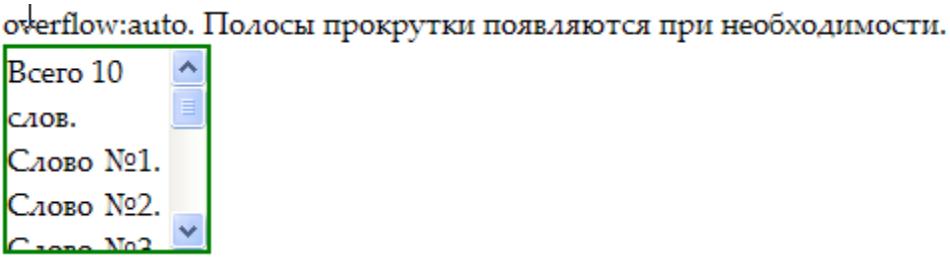


Рисунок 64.

1.3.33. position

Свойство position устанавливает способ позиционирования элемента относительно окна браузера или других объектов на html-странице.

Свойство position может принимать следующие значения:

- absolute – устанавливает абсолютные координаты относительно левого верхнего угла контейнера блока. Координаты указываются с помощью параметров left, right, top и bottom. Абсолютно позиционируемые блоки изымаются из нормального потока. Это значит, что они не влияют на размещение последующих сестринских элементов;
- fixed – устанавливает абсолютные координаты относительно левого верхнего угла окна браузера. Координаты указываются с помощью параметров left, right, top и bottom. При прокрутке содержимого окна браузера элемент разметки не смещается;
- relative – положение элемента устанавливается относительно его исходного места. Добавление атрибутов left, top, right и bottom изменяет позицию элемента и сдвигает его в ту или иную сторону от первоначального расположения, в зависимости от применяемого параметра;
- static – игнорирует установку координат с помощью свойств left, top, right и bottom.

1.3.34. visibility

Свойство visibility указывает браузеру, отображать ему элемент разметки или нет. Свойство может принимать следующие значения:

- visible – отображать;
- hidden – не отображать;
- collapse – если это значение применяется не к строкам или колонкам таблицы, то результат его использования будет таким же, как hidden. В случае использования collapse для содержимого ячеек таблиц, они реагируют, словно к ним было добавлено стилевое свойство display: none. Иными словами, заданные строки и колонки убираются, а таблица перестраивается заново.

1.3.35. Свойства шрифта и текста

font-family

Свойство font-family определяет название шрифта или семейства шрифтов, необходимых для корректного вывода HTML-документа. Например, если в контенте будет встречаться русский текст и математические символы, то необходимо указать, по крайней мере, два шрифта: и для вывода текста, и для вывода формул. Если название шрифта состоит из нескольких слов, то оно должно заключаться в кавычки. Предпочтительнее всегда указывать одновременно несколько шрифтов, разделив их имена запятыми, поскольку на компьютере пользователя может не оказаться того шрифта, в котором был подготовлен документ. В этом случае браузер сможет выбрать тот шрифт из списка, который установлен на компьютере клиента. Кроме того, указывается гарнитура шрифта:

- serif – шрифты с засечками, типа Times;
- sans-serif – рубленые шрифты, типа Arial;
- cursive – курсивные шрифты;
- fantasy – декоративные шрифты;
- monospace – моноширинные шрифты, в которых ширина каждого символа одинаковая.

По умолчанию шрифт Times соответствует значению serif, шрифт Helvetica соответствует значению sans-serif, шрифт Zapf-Chancery соответствует значению cursive, шрифт Western – значению fantasy, шрифт Courier соответствует значению monospase.

Приведем пример (рисунок 65) использования свойства font-family.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
  <body>
    <div style="font-family:Arial, Helvetica,
          sans-serif">Arial, Helvetica, sans-serif
    </div>
    <div style="font-family:'Times New Roman',
          Times, serif">'Times New Roman', Times, serif
    </div>
    <span style="font-family:cursive;">cursive </span>
    <span style="font-family:fantasy;">fantasy</span>
</body>
</html>
```

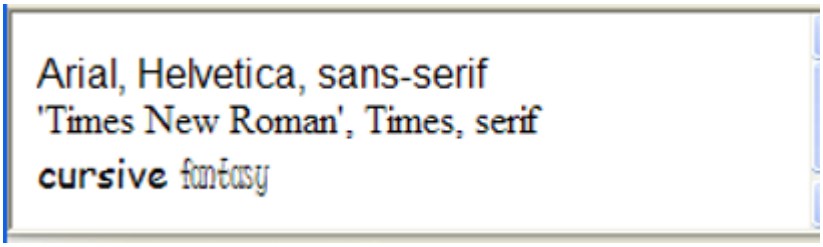


Рисунок 65.

font-size

Свойство font-size управляет размером шрифта. Размер шрифта может быть задан явно. В этом случае он может задаваться в любых единицах определенных CSS, как абсолютных, так и относительных. При указании размера в процентах за 100% берется размер шрифта родительского элемента. Отрицательные значения не допускаются.

Для задания абсолютных размеров шрифта определены семь констант: xx-small, x-small, small, medium, large, x-large, xx-large. Значения этих констант зависят от настроек браузера и операционной системы. Относительные размеры шрифта можно определить с помощью двух констант: larger, smaller. В этом случае размер шрифта текущего элемента определяется относительно размера шрифта родительского элемента.

Приведем пример использования свойства font-size.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Свойство FONT-SIZE</title>
  </head>
  <body>
    <div>Размер по умолчанию</div>
    <div style="font-size:150%">font-size:150%</div>
    <div style="font-size:xx-large">
      font-size:xx-large
    </div>
    <div style="font-size:8px">font-size:8px</div>
    <div style="font-size:1cm">font-size:1cm</div>
  </body>
</html>
```

font-style

Свойство font-style задаёт тип начертания шрифта. Свойство может принимать следующие значения:

normal – задаёт обычное начертание шрифта;

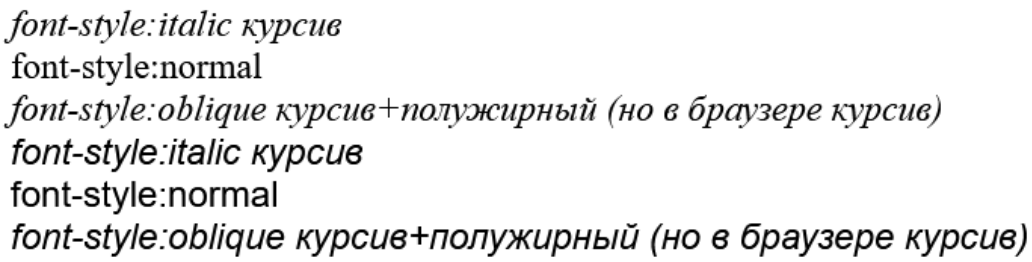
italic – курсивное начертание;

oblique – наклонный шрифт. Наклонный шрифт внешне похож на курсивный, но отличается от него. Курсив – это специальный шрифт, имитирующий рукописный, наклонный шрифт образуется путем наклона символов обычного шрифта вправо.

Приведем пример использования свойства font-style.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Свойство FONT-STYLE</title>
   </head>
   <body>
    <div style="font-style:italic">
      font-style:italic курсив
```

```
    </div>
    <div style="font-style:normal">
      font-style:normal
    </div>
    <div style="font-style:oblique">
      font-style:oblique курсив+полужирный (но в браузере курсив)
    </div>
    <div style="font-style:italic; font-family:fantasy">
      font-style:italic курсив
    </div>
    <div style="font-style:normal; font-family:fantasy">
      font-style:normal
    </div>
    <div style="font-style:oblique; font-family:fantasy">
      font-style:oblique курсив+полужирный (но в браузере
      курсив)
    </div>
  </body>
</html>
```



font-variant

Свойство font-variant определяет, как будут выводиться строчные символы: в виде прописных букв, но меньшего размера, или обычным образом. Способ вывода строчных символов в виде прописных называется капителью. Свойство font-variant определяет выбор варианта шрифта, обладающего двумя наборами знаков (т.е. капителью), и не имеет видимого эффекта для шрифтов, обладающих одним набором знаков. Свойство может принимать следующие допустимые значения:

normal – регистр вывода строчных символов не изменяется (значение по умолчанию);

small-caps – строчные символы выводятся капителью, как заглавные уменьшенного размера.

Приведем пример использования свойства font-variant.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <div style="font-variant:normal">font-variant:normal
      ПРОПИСНЫЕ БУКВЫ. строчные буквы.</div>
    <div style="font-variant:small-caps">font-
      variant:small-caps ПРОПИСНЫЕ БУКВЫ. строчные
```

```
      буквы.</div>
  </body>
</html>
```

font-variant:normal ПРОПИСНЫЕ БУКВЫ. строчные буквы.
FONT-VARIANT:SMALL-CAPS ПРОПИС-НЫЕ БУКВЫ. СТРОЧНЫЕ БУКВЫ.

font-weight

Свойство font-weight устанавливает степень выделения или жирность символов текста при выводе. Свойство может принимать значения:

100, 200, 300, 400, 500, 600, 700, 800, 900 – это условные единицы, каждое значение представляет степень выделения текста по возрастанию. Нормальное начертание шрифта, установленное по умолчанию – 400. Самое светлое начертание шрифта, отображаемое браузером – 100, самое жирное – 900;

- bold – полужирное начертание, соответствует значению 700;
- bolder – более жирное начертание символов, чем в родительском элементе, если в родительском элементе значение font-weight равно 900, то результирующим значением будет также 900;
- lighter – более светлое начертание символов, чем в родительском элементе, если в родительском элементе значение font-weight равно 100, то результирующим значением будет также 100;
- normal – нормальное начертание шрифта принятое по умолчанию.

Приведем пример (рисунок 66) использования свойства font-weight.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>
<div style="font-weight:200">font-weight:200</div>
<div style="font-weight:800">font-weight:800</div>
<div style="font-weight:900">font-weight:900</div>
<div style="font-weight:bold">font-weight:bold</div>
<div style="font-weight:bolder">font-weight:bolder</div>
<div style="font-weight:normal">font-weight:normal</div>
<div style="font-weight:lighter">font-weight:lighter </div>
  </body>
</html>
```

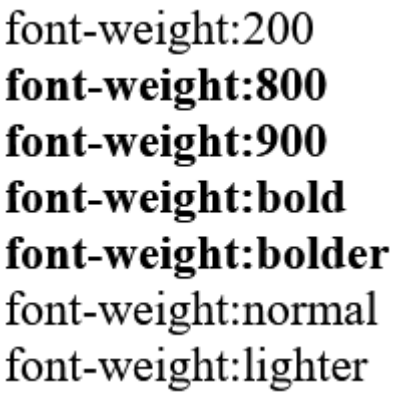


Рисунок 66.

line-height

Свойство line-height устанавливает межстрочный интервал для текста, или расстояние между базовыми линиями соседних строк текста. По умолчанию расстояние между строками зависит от вида и размера шрифта и определяется браузером автоматически, что соответствует значению свойства line-height равному normal. Может задаваться абсолютным значением в любых единицах длины, поддерживаемых CSS. Может также задаваться числом без указания единиц измерения. В этом случае значение воспринимается как множитель от размера шрифта текущего текста. Отрицательное значение межстрочного расстояния не допускается. Может быть задано относительной величиной в процентах. За 100% принимается размера шрифта текущего текста.

font

Свойство font даёт возможность задать сразу шесть свойств шрифта: font-family, font-size, font-style, font-variant, font-weight, и line-height. Значения должны быть перечислены через пробел и могут идти в любом порядке, браузер сам определит, какое из них соответствует нужному свойству. Поскольку, по крайней мере, три параметра допускают числовые значения, следует каждый раз проверять интерпретацию браузера.

letter-spacing

Свойство letter-spacing устанавливает интервал между символами текста. Свойство может принимать значение normal, что означает, что задаётся стандартный интервал между символами текста. Величина этого интервала определяется браузером, исходя из типа шрифта, его размеров и настроек операционной системы. Можно задать расстояние между символами и вручную, указав в свойстве letter-spacing значение в любых абсолютных единицах длины, принятых в CSS, или в относительных единицах основанных на размере шрифта em и ex. Это значение задаёт дополнительное межсимвольное расстояние, которое добавляется к стандартному. Значения могут быть отрицательными. При использовании выравнивания содержимого по ширине для блока браузеры могут не увеличивать или не уменьшать расстояние между символами.

```
Приведем пример использования свойства letter-spacing.
<!doctype html>
<html>
```

```
 < head>
    <meta charset="utf-8">
  </head>
  <body>
    <div style="letter-spacing:normal">
      letter-spacing:normal
    </div>
    <div style="letter-spacing:-0.1em">
      letter-spacing:-0.1em
    </div>
    <div style="letter-spacing:1cm">
      letter-spacing:1cm
    </div>
  </body>
</html>
```

letter-spacing:normal
letter-spacing:-0.1em
l e t t e r - s p a c i n g : 1 c m

text-align

Свойство text-align задаёт тип горизонтального выравнивания текста в пределах блока. Свойство может принимать следующие значения:

- left – выравнивание текста по левому краю браузера или контейнера, в котором он расположен, при этом правый край текста может располагаться лесенкой;
- center – выравнивание текста по центру;
- right – выравнивание текста по правому краю браузера или контейнера, в котором он расположен, при этом левый край текста может располагаться лесенкой;
- justify – выравнивание текста по ширине браузера или контейнера, в котором он расположен, при этом в строку текста могут быть равномерно добавлены пробелы, для того, чтобы и левый, и правый края текста были выровнены.

text-decoration

Свойство text-decoration может добавлять такие эффекты оформления текста как подчеркивание, надстрочное подчеркивание, перечеркивание и мигание. Им отвечают значения свойства underline, overline, line-through и blink соответственно. Значение свойства none отменяет все эффекты оформления, в том числе и подчеркивание ссылок, которое задаётся по умолчанию. Свойство выборочно поддерживается различными браузерами.

text-indent

Свойство text-indent устанавливает величину отступа первой строки блока текста (красная строка). Значение отступа может задаваться в любых единицах длины, принятых в CSS. При задании значения в процентах отступ вычисляется в зависимости от ширины блока. Допускается указывать отрицательные значения. Если значение положительное, то формируется отступ. Если значение отрицательное, то формируется выступ, при этом часть текста может выходить за

рамки блока. Различные браузеры по-разному могут отображать этот кусочек текста, выходящий за рамки блока.

Приведем пример (рисунок 67) использования свойства text-indent.

```
<!doctype html>
  <html>
    <head>
     <meta charset="utf-8">
     <style  >
       div {
         padding-left:50px; background:#CCCCCC;
         width:300px;
         margin-bottom:5px;
         margin-left:20px;}
       div#style1 { text-indent:0 }
       div#style2 { text-indent:-25% }
       div#style3 { text-indent:40px }
     </style>
   </head>
   <body>
     <div>Отступ по умолчанию.</div>
     <div   id="style1">padding-left:50px;   background:#CCCCCC;
width:300px;     margin-bottom:5px;     margin-left:20px;     text-
indent:0</div>
     <div   id="style2">padding-left:50px;   background:#CCCCCC;
width:300px;  margin-bottom:5px;  margin-left:20px;  text-indent:-
25%</div>
     <div   id="style3">padding-left:50px;   background:#CCCCCC;
width:300px;     margin-bottom:5px;     margin-left:20px;     text-
indent:40px</div>
   </body>
</html>
```

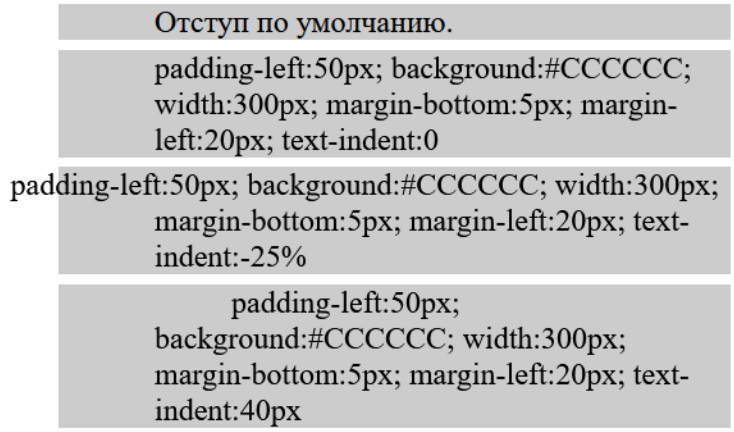


Рисунок 67.

text-transform

Свойство text-transform управляет регистром вывода символов текста. Допустимые значения свойства:

none – не происходит никаких преобразований регистра символов текста в блоке, значение по умолчанию;

capitalize – каждое слово в блоке текста выводится с заглавной буквы;

uppercase – все символы текста преобразуются в прописные;

lowercase – все символы текста преобразуются в строчные.

Приведем пример использования свойства text-transform.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
      div {margin-bottom:4px}
      div.StyleCapitalize{text-transform:capitalize}
      div.StyleLowercase{text-transform:lowercase}
      div.StyleUppercase{text-transform:uppercase}
      span{text-transform:none}
    </style>
  </head>
  <body>
    <div>Отступ по умолчанию.</div>
ТЕКСТ набран БУКВАМИ в РаЗлИчНоМ рЕгИсТрЕ. Вывод по умолчанию.<br>
    <div class="StyleCapitalize">text-transform:capitalize:ТЕКСТ
набран БУКВАМИ в РаЗлИчНоМ рЕгИсТрЕ. <span>text-transform:none:ТЕКСТ
набран БУКВАМИ в РаЗлИчНоМ рЕгИсТрЕ</span></div>
    <div class="StyleLowercase">text-transform:lowercase:ТЕКСТ
набран БУКВАМИ в РаЗлИчНоМ рЕгИсТрЕ.</div>
    <div class="StyleUppercase">text-transform:uppercase:ТЕКСТ
набран БУКВАМИ в РаЗлИчНоМ рЕгИсТрЕ.</div>
  </body>
</html>
```

Отступ по умолчанию.
ТЕКСТ набран БУКВАМИ в РаЗлИчНоМ рЕгИсТрЕ. Вывод по умолчанию.
Text-transform:capitalize:ТЕКСТ Набран БУКВАМИ В РаЗлИчНоМ РЕгИсТрЕ. text-transform:none:ТЕКСТ набран БУКВАМИ в РаЗлИчНоМ рЕгИсТрЕ
text-transform:lowercase:текст набран буквами в различном регистре.
TEXT-TRANSFORM:UPPERCASE:ТЕКСТ НАБРАН БУКВАМИ В РАЗЛИЧНОМ РЕГИСТРЕ.

В приведенном примере в первом блоке текста осуществлен вывод текста по умолчанию. Как текст был написан в html-документе, так он и выводится браузером. Во втором блоке было применено свойство `text-transform:capitalize`. Каждое слово, не зависимо от того, как оно было записано в html-документе, выводится с заглавной буквы. Внутри второго блока, в рамках тега , были отменены установки для вывода каждого слова текста с заглавной буквы, поэтому текст снова был выведен так же, как и набран. В третьем блоке было применено свойство `text-transform:lowercase`, что привело к тому, что весь текст был отображен браузером строчными буквами. Для четвертого блока текста было установлено `text-transform:uppercase`, в результате чего все содержимое блока было выведено браузером заглавными буквами.

vertical-align

Свойство vertical-align управляет вертикальным выравниванием содержимого блока относительно родительского блока или окружающего теста. Свойство может принимать следующие значения:

- baseline – осуществляется выравнивание базисной линий блока относительно базисной линии содержащего его родительского блока, если у бло-

ка нет базисной линии, то выполняется выравнивание его нижней границы относительно базисной линии родительского блока;

- sub – базисная линия блока смещается вниз до уровня нижнего индекса родительского блока, в результате чего блок отображается в виде нижнего индекса, при этом размер шрифта текста не изменяется;
- super – базисная линия блока смещается вверх до уровня верхнего индекса родительского блока, в результате чего блок отображается в виде верхнего индекса, при этом размер шрифта текста не изменяется;
- top – происходит выравнивание верхнего края блока по верхней границе родительского элемента (строки);
- text-top – происходит выравнивание верхнего края блока по верху самого высокого элемента строки родительского блока;
- middle – осуществляется выравнивание средней по вертикали точки блока относительно суммы уровня базисной линии родительского блока и половины высоты родительского блока;
- bottom – происходит выравнивание нижнего края блока по нижней границе родительского элемента (строки);
- text-bottom – происходит выравнивание нижнего края блока по нижней кромке шрифта родительского блока;
- числовое значение% – осуществляется смещение блока на заданную величину, вычисляемую как процент от значения свойства line-height, вверх при положительном значении или вниз при отрицательном значении. Задание 0% аналогично значению baseline;

числовое значение – осуществляется смещение блока на заданную величину, вверх при положительном значении или вниз при отрицательном значении. Значение 0 равносильно значению baseline.

white-space

Свойство white-space определяет тип обработки идущих подряд пробельных символов внутри текста. Допускаются следующие значения свойства:

normal – пробельные символы обрабатываются стандартным образом, т. е. любая последовательность подряд стоящих пробелов, символов табуляции и пустых строк отображается браузером как один пробел, если речь идет не о содержимом тега <pre>;

pre – последовательность подряд стоящих пробелов, символов табуляции и пустых строк отображается браузером без преобразований, так как она была задана в коде исходного html-документа;

nowrap – пробельные символы обрабатываются стандартным образом, как при значении normal, но запрещен перенос слов.

Приведем пример (рисунок 68) использования свойства white-space.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
```

```
        div {
             width:400px;
             background:#99CCFF;
             margin-bottom:4px;}
        div.classNormal{white-space:normal}
        div.classNowrap{white-space:nowrap}
        div.classPre{white-space:pre}
      </style>
    </head>
    <body>
      <div class="classNormal">Обработка пробелов
        обычным образом.       Несколько
                        подряд
 идущих пробелов заменяются
 одним, при необходимости происходит автоматический перенос текста на
новую строку.
      </div>
      <div class="classNowrap">Вывод текста       без

 переносов. Вывод текста       без переносов. Вывод текста без
переносов.
    </div>
    <div class="classPre">
 Вывод
  текста
   в
    авторской
     редакции.</div>
    </body>
 </html>
```

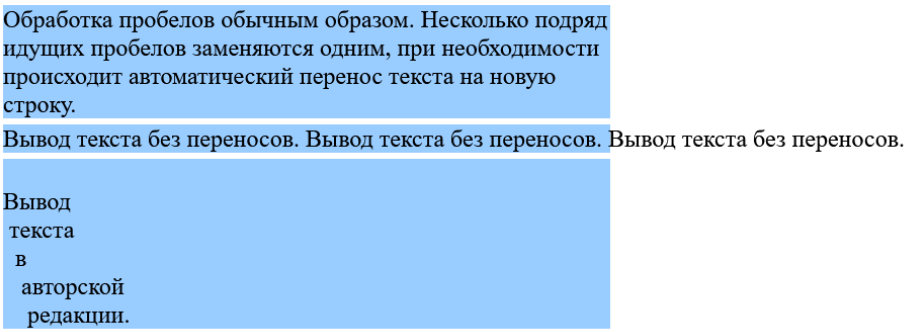


Рисунок 68.

Из приведенного примера видно, что в первом блоке, к которому применено свойство white-space со значением normal, несколько подряд идущих пробельных символов трактуются браузером как один разделитель, а также, поскольку установлена фиксированная ширина блока, происходит автоматический перенос слов на новую строку по достижении текстом правой границы блока.

Во втором блоке свойство white-space установлено в nowrap, что приводит к выводу текста в одну строку.

В третьем блоке применено свойство white-space со значением pre, что эквивалентно применению тега <pre>, и текст выводится в авторском представлении.

word-spacing

Свойство word-spacing устанавливает величину пробельного символа или расстояние между словами в тексте. Свойство может принимать следующие значения:

normal – величина пробела стандартная, и зависит от выбранного шрифта и настроек браузера;

числовое значение – величина пробела задаётся напрямую, как сумма стандартной величины и заданного значения, числовое значение может указываться в любых единицах принятых в CSS, кроме процентов. Числовые значения могут быть и отрицательными, но их отображение зависит от браузера.

Если параметр, управляющий выравниванием текста text-align, установлен в значение justify, то величина расстояния между отдельными словами может быть больше значения указанного в свойстве word-spacing, поскольку интервал между словами подбирается и устанавливается браузером принудительно таким образом, чтобы строка текста была выровнена по ширине родительского блока.

Приведем пример использования свойства word-spacing.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
     .WordSpacingNormal{word-spacing:normal}
     .WordSpacing10px{word-spacing:10px}
     .WordSpacing7em{word-spacing:7em}
     .WordSpacingNegativ3{word-spacing:-3px}
     .WordSpacingNegativ30{word-spacing:-30px}
     #AlignJustgify{text-align:justify}
    </style>
  </head>
  <body>
    <div>Расстояние между словами по умолчанию.</div>
    <div class="WordSpacingNormal">WordSpacingNormal WordSpac-
ingNormal</div>
    <div class="WordSpacing7em">WordSpacing7em WordSpac-ing7em</div>
    <div class="WordSpacing10px">WordSpacing10px WordSpac-
ing10px</div>
    <div class="WordSpacingNegativ3">WordSpacingNegativ3 WordSpac-
ingNegativ3</div>
    <div class="WordSpacingNegativ30">WordSpacingNegativ30 Word-
SpacingNegativ30</div>
    <div class="WordSpacing10px" id="AlignJustgify">text-
align:justify text-align:justify</div>
  </body>
</html>
```

Расстояние между словами по умолчанию.
WordSpacingNormal WordSpac-ingNormal
WordSpacing7em WordSpac-ing7em
WordSpacing10px WordSpac-ing10px
WordSpacingNegativ3 WordSpac-ingNegativ3
WordSpacingNegativ30 Word-SpacingNegativ30
text-align:justify text-align:justify text-align:justify text-align:justify text-align:justify
text-align:justify text-align:justify

В примере для второй строки текста установлено значение свойства `word-spacing:normal`. Это эквивалентно приданию пробелу стандартной величины, такой же, как и при выводе текста без заданных значений свойства word-spacing в первой строке.

В третьей и четвертой строках устанавливается пробел в `7em` и `10px` соответственно. В пятой и шестой строках применяются отрицательные значения свойства word-spacing. При небольшом значении в `-3px` это приводит к более сжатому выводу текста в пятой строке. Отрицательное значение свойства `-30px` приводит к потере читабельности текста и наползанию слов друг на друга в шестой строке.

Содержимое последнего блока, занимающего две последних строки текста, выравнивается по ширине страницы, поэтому величина пробела между словами различна в седьмой и восьмой строках.

1.3.36. Свойства списков

list-style-image

Свойство list-style-image устанавливает произвольное изображение в качестве маркера списка. Атрибут наследуется, поэтому для отдельных элементов списка для восстановления маркера используется значение none. Значением свойства является адрес файла с изображением маркера. Аргумент none отменяет изображение в качестве маркера для родительского элемента.

Приведем пример использования свойства *list-style-image*.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <ul style="list-style-image:url(image/bullet.GIF)">
    <li>Иванов</li>
      <li>Петров</li>
      <li>Сидоров</li>
    </ul>
</body>
</html>
```

list-style-position

Свойство list-style-position устанавливает, где должен находиться маркер по отношению к элементу списка. Возможные значения свойства:

- inside – внутри блока элемента списка;
- outside – снаружи, перед блоком элемента списка.

list-style-type

Свойство list-style-type отвечает за установку типа маркера элементов списка. Свойство может принимать следующие значения:

- none – маркер списка отсутствует;
- disc – маркер списка закрашенная окружность;
- circle – маркер списка не закрашенная окружность;
- square – маркер списка квадрат;
- decimal – список промаркирован арабскими цифрами;
- lower-roman – список промаркирован маленькими римскими цифрами;
- upper-roman – список промаркирован большими римскими цифрами;
- lower-alpha – список промаркирован малыми латинскими буквами;
- upper-alpha – список промаркирован большими латинскими буквами.

list-style

Свойство list-style является сокращенным вариантом одновременного использования свойств list-style-image, list-style-position, list-style-type.

1.3.37. Свойства таблиц

caption-side

Свойство caption-side используется для установки позиции вывода заголовка таблицы, который определен в теге <caption>. Допустимы следующие значения свойства:

- top – поле заголовка располагается над таблицей;
- bottom – поле заголовка располагается под таблицей;
- initial – значение по умолчанию;
- inherit – поле заголовка располагается как у родительского элемента.
- border-collapse

Свойство border-collapse определяет модель вывода границ таблицы. Это свойство применяется, если установлен вывод рамки для ячеек таблицы и для самой таблицы. Свойство может принимать следующие значения: separate, collapse, initial, inherit.

- separate – для каждой ячейки отображается своя граница, поэтому в местах стыка двух ячеек или ячейки и границы таблицы выводится линия двойной толщины. Значение по умолчанию;
- collapse – для каждой ячейки отображается своя граница, но в местах стыка двух ячеек или ячейки и границы таблицы выводится одна линия.

При использовании значения separate для border-collapse можно задать расстояние между ячейками с помощью border-spacing.

```
table {  border-collapse: separate;  border-spacing: 10px;}
```

Приведем пример (рисунок 69) использования свойства border-collapse.

```
<!doctype html>
```

```
<html>
  <head>
    <meta charset="utf-8">
    <style>
      table,th { border:2px solid blue }
      td{ border:2px solid red }
      table { margin-bottom:10px }
      table.ClassCollapse { border-collapse:collapse }
    </style>
  </head>
  <body>
    <table>
      <tr>
        <th>ФИО</th>        <th>Средний балл</th>
      </tr>
      <tr>
        <td>Иванов И.И.</td>    <td>7.5</td>
      </tr>
      <tr>
        <td>Петров П.П.</td>    <td>8.4</td>
      </tr>
    </table>

    <table class="ClassCollapse">
      <tr>
        <th>ФИО</th>     <th>Средний балл</th>
      </tr>
      <tr>
        <td>Иванов И.И.</td>    <td>7.5</td>
      </tr>
      <tr>
        <td>Петров П.П.</td>    <td>8.4</td>
      </tr>
    </table>
  </body>
</html>
```

ФИО	Средний балл
Иванов И.И.	7.5
Петров П.П.	8.4

ФИО	Средний балл
Иванов И.И.	7.5
Петров П.П.	8.4

Рисунок 69.

В нижней таблице определено `border-collapse:collapse`, что заставляет браузер на стыке двух границ выводить одну линию. При этом при различном значении параметра границы у разных блоков значение этого же параметра для общей границы выбирается равным значению параметра родительского элемента. Например, цвет границы ячеек таблицы установлен красный, а

цвет заголовочных ячеек и цвет границы самой таблицы – синий. На стыке границ ячеек и таблицы за цвет границы принимается синий, цвет границы всей таблицы.

empty-cells

По умолчанию пустые ячейки таблицы не отображаются, для вывода ячейки таблицы, не содержащей никакой информации, в нее можно поместить не отображаемый символ неразрывного пробела .

Используя свойство empty-cells можно выводить (show) или не выводить (hide) пустые ячейки не содержащие символа неразрывного пробела (рисунок 70). Если свойство empty-cells установлено в hide, то пустые ячейки не отображаются.

ФИО	Средний балл
Иванов И.И.	7.5
Петров П.П.	8.4
Итого	

Рисунок 70.

table-layout

Свойство table-layout определяет алгоритм, по которому браузер будет определять размеры ячеек таблицы при ее выводе. Возможные значения свойства:

- auto – браузер загружает всю таблицу, анализирует ее, определяет размер ячеек (высота и ширина ячеек определяется в зависимости от их содержимого), и только после этого отображает всю таблицу;
- fixed – браузер начинает вывод таблицы сразу, при этом ширина колонок определяется либо с помощью тега <col>, либо вычисляется на основе первой строки. Если данные о форматировании первой строки таблицы не заданы явно, то в этом случае таблица по ширине занимает все свободное место родительского элемента и делится на колонки равной ширины. Если при фиксированной ширине колонок, их содержимое не помещается в ячейку заданной ширины, то оно либо частично не будет видно, либо наложится поверх соседней ячейки в зависимости от используемого браузера.

Пример использования (рисунок 71) свойства table-layout.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
      table,td,th { border:2px solid blue }
      table{
            margin-bottom:10px;
            border-collapse:collapse;
            background-color:#FFCCFF; }
```

```
        .StLayoutFixed { table-layout:fixed }
        .FixedWidth30 {width:30px }
        .FixedWidth15 {width:150px }
        .FixedWidth25 {width:250px }
     </style>
   </head>
   <body>
   <!--Размеры ячеек устанавливаются исходя из их содержимого, по
умолчанию. -->
   <table>
     <tr><th>ФИО</th><th>Средний балл</th></tr>
     <tr><td>Иванов И.И.</td><td>7.5</td></tr>
     <tr><td>Петров П.П.</td><td>8.4</td></tr>
   </table>
   <!--Ширина ячеек устанавливается одинаковой. -->
   <table class="StLayoutFixed">
     <tr><th>ФИО</th><th>Средний балл</th></tr>
     <tr><td>Иванов И.И.</td><td>7.5</td></tr>
     <tr><td>Петров П.П.</td><td>8.4</td></tr>
   </table>
   <!--Размеры ячеек устанавливаются по размеру ячеек заголовка,
установленным явно. -->
   <table class="StLayoutFixed">
     <tr>
       <th  class="FixedWidth25">ФИО</th>
       <th  class="FixedWidth15">Средний балл</th>
     </tr>
     <tr><td>Иванов И.И.</td><td>7.5</td></tr>
     <tr><td>Петров П.П.</td><td>8.4</td></tr>
   </table>
   <!--Размеры ячеек устанавливаются по размеру ячеек заголовка,
установленным явно, даже если содержимое ячеек отображается не
полностью. -->
   <table class="StLayoutFixed">
     <tr>
       <th  class="FixedWidth30">ФИО</th>
       <th  class="FixedWidth15">Средний балл</th> </tr>
       <tr><td>Иванов И.И.</td><td>7.5</td></tr>
       <tr><td>Петров П.П.</td><td>8.4</td></tr>
   </table>
   <!--Размеры ячеек первого столбца устанавливаются по размеру
установленному явно в теге COL. -->
   <table class="StLayoutFixed">
     <col width="150">
     <tr><th>ФИО</th><th>Средний балл</th> </tr>
     <tr><td>Иванов И.И.</td><td>7.5</td></tr>
     <tr><td>Петров П.П.</td><td>8.4</td></tr>
   </table>
   </body>
 </html>
```

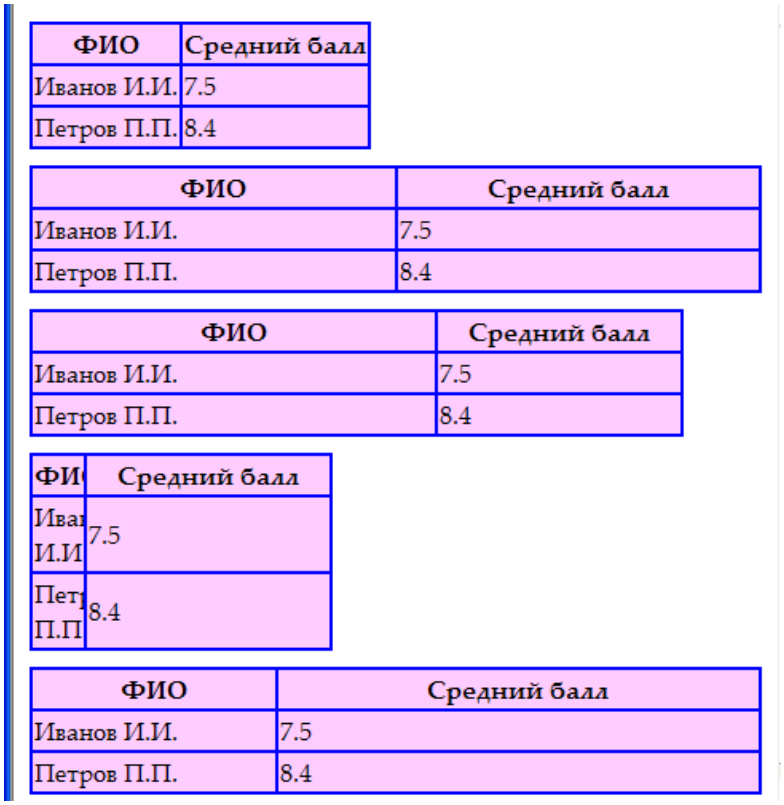


Рисунок 71.

В первой из выводимых таблиц размеры ячеек устанавливаются автоматически исходя из их содержимого, по умолчанию.

Во второй таблице свойство table-layout установлено в fixed, но ширина не задана явно, поэтому таблица заняла по ширине все пространство окна, а ширина столбцов взята одинаковой.

В третьей таблице свойство table-layout установлено в fixed, размеры ячеек устанавливаются по размеру ячеек заголовка, установленным явно.

В четвертой таблице свойство table-layout установлено в fixed, и размеры ячеек также устанавливаются по размеру ячеек заголовка, установленным явно, даже если содержимое ячеек первого столбца видно только частично. Браузер обрезал их содержимое при выводе.

В пятой таблице свойство table-layout установлено в fixed, при этом ширина первого столбца устанавливается по размеру установленному явно в теге <col>, а под второй столбец отводится все пространство до правого края родительского элемента (окна браузера).

1.3.38. Свойство cursor

Устанавливает вид курсора мыши, когда он находится в пределах элемента разметки. Вид курсора по умолчанию зависит от операционной системы и установленных параметров. Свойство cursor может принимать следующие значения:

- default – курсор, используемый для данной операционной системы по умолчанию, часто представляется в виде большой стрелки;
- crosshair – курсор-прицел в виде крестика;

- help – курсор, означающий, что для объекта, на который он указывает, имеется справочная информация, часто представляется в виде вопросительного знака или воздушного шара;
- move – курсор, определяющий объект, который можно переместить, пересечение горизонтальной и вертикальной двунаправленных стрелок;
- pointer – курсор представляется указателем, обозначающим ссылку, обычно в виде указывающей руки;
- progress – курсор, указывающий на выполнение программы, обычно в виде стрелки с песочными часами;
- text – курсор, используемый при выделении текста, часто представляется в виде вертикальной линии ограниченной короткими горизонтальными линиями сверху и снизу;
- wait – курсор, указывающий на занятость программы, часто представляется в виде песочных часов или циферблата;
- e-resize, ne-resize, nw-resize, n-resize, se-resize, sw-resize, s-resize, w-resize – курсоры, определяющие перемещение некоторого края, обычно в виде двунаправленной стрелки соответствующей ориентации;
- <uri> – браузер загружает курсор из ресурса, задаваемого этим URI. Значение состоит из трех частей: <uri> базового курсора, <uri> резервного курсора, стандартный курсор браузера. Если браузеру не удаётся загрузить курсор, расположенный первым в списке курсоров, он должен попытаться загрузить резервный. Если браузеру не удаётся обработать ни одного курсора, заданного пользователем, он должен использовать общий курсор, расположенный в конце этого списка.

1.3.39. Свойство quotes

Свойство quotes устанавливает тип открывающей и закрывающей кавычек, который применяется в тексте документа для выделения содержимого тега ***<q>***. Тип кавычек может быть задан символами, символами юникода или специальными символами HTML.

Тип	Спецкод HTML	Символ юникода	Описание
"	&#34;	\0022	Двойная кавычка, применяется обычно для обозначения символа дюйма.
'	&#39;	\0027	Апостроф.
«	« или «	\00AB	Открывающая двойная угловая кавычка.
»	» или »	\00BB	Закрывающая двойная угловая кавычка.
‘	&#8216;	\2018	Открывающая одинарная кавычка.
’	&#8217;	\2019	Закрывающая одинарная кавычка.

“	“	\201C	Открывающая двойная кавычка в англоязычных текстах или закрывающая для русского языка.
”	”	\201D	Закрывающая двойная кавычка в англоязычных текстах.
„	„	\201E	Открывающая двойная кавычка в русскоязычных текстах.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
      .s1{ quotes:"\00ab" "\00bb" }
      .s2{ quotes:"\2018" "\2019" }
      .s3{ quotes:"\201c" "\201d" }
    </style>
  </head>
  <body>
 Просто текст. <q class="s1">Это цитата. <q class="s2">А это цитата в цитате.</q> Продолжение цитаты.</q> Снова просто текст.<br>
 <p class="s3">А в этом абзаце <q style="quotes: '<' '>'">свое</q> выделение цитат. <q>Это цитата.</q></p>
  </body>
</html>
```

В приведенном примере определено три стиля для выделения цитат во внедренной таблице стилей и один стиль во встроенной таблице стилей.

1.3.40. Свойство z-index

Свойство z-index определяет порядок отображения элементов разметки. Свойство может принимать значение auto, в этом случае порядок перекрытия элементов задаётся по умолчанию: элементы, определенные в html-коде раньше, перекрываются заданными позже. Кроме константы auto значением свойства может быть любое целое число, оно и является индексом в последовательности отображения элементов. Элементы с большим значением индекса будут перекрывать элементы с меньшим значением. Спецификация разрешает использовать отрицательные значения z-index**.**

1.3.41. Свойство zoom

Свойство zoom управляет масштабом вывода элементов разметки. Значением свойства может быть число с плавающей точкой, задающее множитель увеличения или уменьшения, также свойство может быть задано в процентах или константой normal. Значение normal является значением по умолчанию и задаёт масштаб 1.0 или 100%.

1.3.42. Псевдоклассы гипертекстовых ссылок

Псевдоклассы гипертекстовых ссылок управляют отображением гипертекстовых ссылок и применяются только к тегу <a>**.** Обычно браузеры различным образом отображают просмотренные и не просмотренные ссылки, ссылки находящиеся в фокусе и ссылки в момент их активации.

Псевдокласс :link применяется для описания стиля отображения не посещённых ссылок.

Псевдокласс :visited применяется для описания стиля отображения ссылок, которые уже посещались.

Псевдокласс :hover применяется для описания стиля отображения ссылок, в момент, когда они находятся в фокусе, но не активизированы. Например, браузер может применять этот псевдокласс, когда указатель мыши находится над ссылкой, но щелчка мыши не произошло.

Псевдокласс :active применяется, в момент активизации ссылки, а именно, между моментами, когда пользователь нажимает кнопку мыши и отпускает ее.

Браузеры не обязаны проводить новое форматирование html-документа во время выбора ссылок и переходов по ссылкам. Например, в таблице стилей может быть указано, что размер шрифта у посещённой ссылки больше, чем у не посещённой. Поскольку это может привести к изменению положения текста при возвращении в данную точку документа после посещения ссылки, браузер может проигнорировать соответствующее стилевое правило. Псевдоклассы гипертекстовых ссылок по-разному поддерживаются различными браузерами. При описании псевдоклассов гипертекстовых ссылок важен порядок их описания. Правило для :hover должно располагаться после правил :link и :visited, так как в противном случае правила каскадирования скроют свойства правила :hover. Аналогично :active должно находиться после :hover, т. к. когда пользователь устанавливает указатель поверх ссылки, то одновременно активизирует её. Поскольку описанные псевдоклассы управляют только отображением гипертекстовых ссылок и применяются только к тегу <a>, то в стилевой таблице можно явно не указывать название тега. Следующие два правила эквивалентны.

```
:visited {color:silver}
a:visited {color:silver}
```

1.3.43. Псевдоэлементы :ferst-letter и :first-line

:first-letter

Псевдоэлемент :first-letter используется для форматирования первого символа элемента разметки для создания типографских эффектов заглавной буквы и буквицы.

Эффект заглавной буквы достигается, если свойство float установлено в none (значение по умолчанию). В этом случае первая буква ведет себя как элемент строки, т. е. находится на одной линии с другими символами строки. Если значение свойства float изменено, то первая буква будет вести себя как перемещаемый объект и можно добиться эффекта буквицы.

В разных браузерах эффект применения псевдоэлемента *:first-letter* может быть различным.

:first-line

Псевдоэлемент :first-line используется для изменения стиля первой строки абзаца. Он может применяться только к элементам уровня блока.

Браузеры по-разному интерпретируют инструкции описанные в :first-line.

1.3.44. Анимирование с помощью CSS

CSS-анимации состоят из двух компонентов: стилевое описание анимации и, возможно, набор ключевых кадров, определяющих начальное, конечное и, при желании, промежуточное состояние анимируемых стилей.

Существует 4 свойства для описания CSS-переходов между двумя состояниями элемента:

- transition-property – изменение какого именно свойства необходимо анимировать (можно указать all)
- transition-duration – продолжительность перехода
- transition-timing-function –функция перехода. Она описывает, как процесс анимации будет распределён во времени.
- transition-delay – задержка перед началом анимации.

Имеется также универсальное свойство transition, которое позволяет анимировать несколько свойств одновременно и задать их одновременно в последовательности: property duration timing-function delay.

С помощью свойства transform можно трансформировать элементы различными способами. Значения transform:

Функция	Описание
translate(x,y)	Смещает элемент от изначальной позиции по горизонтали и вертикали.
translateX(x)	Смещает элемент по горизонтали.
translateY(y)	Смещает элемент по вертикали.
scale(x,y)	Растягивает элемент по вертикали и горизонтали.
scaleX(x)	Растягивает элемент по горизонтали.
scaleY(y)	Растягивает элемент по вертикали.
rotate(градусы)	Поворачивает элемент по часовой стрелке.
skew(x,y)	Скашивает элемент по горизонтали и вертикали.
skewX(x)	Скашивает элемент по горизонтали.
skewY(y)	Скашивает элемент по вертикали.

Пример:

```
Transition: .pic {
    margin-left: 80%;
```

```
    transform: rotate(-30deg) scale(1.5);
    transition: all 2s ease-in;
}
```

Можно объединить несколько анимаций вместе, используя @keyframes. Оно представляет собой контейнер для которого задаётся «имя» анимации, а внутри него помещаются свойства оформления – правила: что, когда и где анимировать. После этого можно использовать свойство animation, чтобы назначить анимацию на элемент и определить её дополнительные параметры.

```
@keyframes anim { /* даём имя анимации: "anim" */
from {margin-left:3px;}
to {margin-left:500px;}
}

#wrap {
animation: anim 4s infinite alternate;
/* имя анимации, продолжительность, сколько раз,
менять направление анимации каждый раз (alternate) */
}
```

1.3.45. Библиотеки, используемые при создании сайтов

Пользователь в первую очередь взаимодействует с интерфейсом, и именно от его качества зависит первое впечатление. Чтобы не писать CSS "с нуля" и не изобретать велосипед, разработчики используют библиотеки стилей — готовые наборы CSS-классов и компонентов, которые можно применять к HTML-элементам.

Преимущества использования библиотек стилей:

Скорость разработки: не нужно писать стили вручную — достаточно подключить библиотеку и использовать готовые классы.

Адаптивность: большинство библиотек уже включают поддержку мобильных устройств.

Единый стиль: элементы интерфейса выглядят согласованно и профессионально.

Кроссбраузерность: стили корректно отображаются в разных браузерах. Библиотеки широко используются и проверены временем, что снижает вероятность багов.

Bootstrap: универсальный инструмент для создания адаптивных сайтов

Bootstrap — это один из самых популярных фреймворков для фронтенд-разработки, созданный в 2011 году разработчиками из Twitter. Он представляет собой набор готовых CSS- и JavaScript-компонентов, предназначенных для быстрой и удобной верстки веб-страниц. Основная цель Bootstrap — упростить процесс создания адаптивных, кроссбраузерных и визуально привлекательных интерфейсов.

Bootstrap включает в себя:

Систему сетки (Grid system) для построения макета;

Готовые UI-компоненты: кнопки, формы, карточки, модальные окна, навигационные панели и др.;

Утилитарные классы для управления отступами, цветами, выравниванием, размерами и т.д.;

JavaScript-плагины, реализующие интерактивные элементы: выпадающие списки, карусели, всплывающие подсказки и модальные окна.

Принцип Mobile First

Одна из ключевых особенностей Bootstrap — это подход Mobile First. Это означает, что стили и компоненты изначально разрабатываются с учётом отображения на мобильных устройствах, а затем масштабируются для планшетов и десктопов. Такой подход позволяет создавать сайты, которые одинаково хорошо выглядят на экранах любого размера.

Система сетки (Grid System)

Bootstrap использует 12-колоночную систему сетки, основанную на Flexbox. Это позволяет легко создавать сложные макеты, адаптирующиеся под разные разрешения экрана. Сетка делится на ряды (.row) и колонки (.col-*), которые можно комбинировать и настраивать с помощью медиазапросов.

```
html
<div class="container">
  <div class="row">
    <div class="col-md-6">Левая колонка</div>
    <div class="col-md-6">Правая колонка</div>
  </div>
</div>
```

В этом примере две колонки занимают по 6 из 12 возможных частей на экранах среднего размера и выше.

Компоненты интерфейса

Bootstrap предоставляет десятки готовых компонентов, которые можно использовать без дополнительной стилизации. Эти компоненты легко настраиваются и комбинируются между собой:

Кнопки (.btn, .btn-primary, .btn-danger и др.)

Формы с валидацией и адаптивной верстkой

Карточки (.card) — универсальные контейнеры для контента

Модальные окна (.modal) — всплывающие диалоговые окна

Навигационные панели (.navbar) — верхние и боковые меню

Утилиты и кастомизация

Bootstrap содержит множество утилитарных классов, которые позволяют быстро управлять внешним видом элементов:

Отступы: m-3, p-2, mt-4, mb-0
Цвета: text-primary, bg-success, border-danger
Выравнивание: text-center, d-flex, justify-content-between

Кроме того, Bootstrap можно кастомизировать с помощью переменных SCSS. Это позволяет изменить цветовую палитру, размеры шрифтов, отступы и другие параметры под нужды конкретного проекта.

JavaScript-компоненты
Bootstrap включает в себя набор JavaScript-плагинов, основанных на библиотеке Popper.js.. Они позволяют реализовать интерактивные элементы:
Dropdown — выпадающие списки
Collapse — скрытие и раскрытие блоков
Carousel — слайдеры изображений
Tooltip и Popover — всплывающие подсказки
Эти компоненты можно использовать как с помощью HTML-атрибутов (data-bs-*), так и через JavaScript API.

Преимущества и недостатки
Плюсы:
Быстрый старт и высокая скорость разработки
Адаптивность и кроссбраузерность "из коробки"
Большое сообщество и обилие документации
Совместимость с большинством современных фреймворков (React, Angular, Vue)

Минусы:
Избыточность кода при использовании только части функционала
Ограниченная гибкость без кастомизации
Возможность "однотипного" внешнего вида сайтов.

Tailwind CSS — утилитарный CSS-фреймворк нового поколения

Tailwind CSS — это современная CSS-библиотека, построенная на принципе утилитарного подхода. В отличие от традиционных фреймворков, Tailwind не предлагает готовых компонентов, а предоставляет разработчику набор низкоуровневых классов, каждый из которых отвечает за конкретное визуальное свойство: отступ, цвет, размер, выравнивание и т.д.

Tailwind CSS особенно популярен среди разработчиков, создающих кастомные интерфейсы, где важна точность, гибкость и соответствие дизайн-макету. Он идеально подходит для проектов, где внешний вид должен строго соответствовать требованиям дизайнера.

Принцип работы

Tailwind CSS позволяет "собирать" внешний вид элементов прямо в HTML, используя классы вроде:

p-4 — внутренний отступ 1rem

bg-blue-500 — фон синего цвета

text-center — выравнивание текста по центру

rounded-lg — скругление углов

Такой подход позволяет быстро прототипировать интерфейсы и точно контролировать стили без написания отдельного CSS. Это особенно удобно при работе с компонентными фреймворками, такими как React, Vue или Svelte, где каждый элемент — это отдельный блок с собственным стилем.

Tailwind CSS также поддерживает псевдоклассы (hover:, focus:, active:), состояния (disabled:), а также условные стили для адаптивной верстки (sm:, md:, lg:), что делает его мощным инструментом для создания сложных интерфейсов.

Адаптивность и медиазапросы

Tailwind поддерживает адаптивную верстку с помощью префиксов:

sm: — для маленьких экранов

md: — для средних

lg: — для больших

xl: — для очень больших

Пример:

```
html
<div class="text-sm md:text-lg lg:text-xl">
  Адаптивный текст
</div>
```

Здесь размер текста будет меняться в зависимости от ширины экрана, что позволяет создавать гибкие и отзывчивые интерфейсы без написания медиазапросов вручную.

Кастомизация и конфигурация

Tailwind поставляется с конфигурационным файлом tailwind.config.js, в котором можно:

Добавлять собственные цвета, шрифты, размеры

Настраивать брейкпоинты

Создавать собственные утилиты и компоненты

Включать или отключать определённые классы для оптимизации размера итогового CSS

Tailwind также поддерживает JIT-компиляцию (Just-In-Time), которая генерирует CSS-классы "на лету" только для тех, что реально используются в проекте. Это позволяет значительно уменьшить размер итогового файла стилей и ускорить загрузку страницы.

Преимущества
Полный контроль над стилем
Отличная интеграция с React, Vue, Svelte
Быстрая разработка без переключения между HTML и CSS
Поддержка JIT-компиляции
Гибкость при создании дизайн-систем

Недостатки
HTML может быть перегружен классами, что затрудняет чтение
Требует времени на освоение синтаксиса
Без предварительной настройки может быть громоздким
Не предоставляет готовых компонентов — всё нужно собирать вручную

Bulma — чистый CSS-фреймворк на основе Flexbox

Bulma — это легковесный CSS-фреймворк, построенный полностью на Flexbox. В отличие от Bootstrap, он не использует JavaScript, что делает его простым в подключении и использовании. Bulma ориентирован на чистую, семантичную верстку и предоставляет разработчику удобные классы для построения адаптивных интерфейсов.

Отлично подходит для образовательных проектов и быстрого прототипирования. Нет зависимости от JavaScript — только CSS. Благодаря своей простоте, он часто используется в учебных проектах, хакатонах и MVP-прототипах. Он также хорошо подходит для начинающих разработчиков, которые только осваивают основы верстки.

Bulma — это отличный выбор для тех, кто хочет быстро создать адаптивный сайт без лишней сложности. Он особенно полезен в образовательной среде, где важно сосредоточиться на структуре и семантике HTML, не отвлекаясь на JavaScript.

Основные компоненты Bulma:
Сетку (columns, column) — построение макета с помощью Flexbox
Кнопки (button, is-primary, is-danger)
Формы (input, textarea, select)
Карточки, панели, таблицы, навигацию

Пример:

```
html
<div class="columns">
  <div class="column is-half">
    Левая часть
  </div>
  <div class="column is-half">
    Правая часть
```

```
    </div>
  </div>
```

Сетка в Bulma интуитивно понятна и легко масштабируется. Она автоматически адаптируется под размер экрана, а классы вроде is-one-third, is-three-quarters позволяют точно управлять шириной колонок.

Особенности
Семантичные классы: is-success, is-warning, is-centered
Простая структура: легко читаемый HTML
Поддержка адаптивности без сложных медиазапросов

Bulma также поддерживает модификаторы для управления цветами, размерами, выравниванием и состояниями элементов. Например, is-hovered, is-focused, is-loading — всё это можно применять к кнопкам и другим компонентам.

Преимущества
Чистый HTML без перегрузки
Быстрое освоение для новичков
Отличная поддержка Flexbox
Нет зависимости от JavaScript
Хорошо подходит для небольших и средних проектов

Недостатки
Меньше компонентов, чем у Bootstrap
Нет встроенных JavaScript-элементов
Ограниченная кастомизация без SCSS
Не подходит для сложных интерфейсов без дополнительной настройки

1.4. Язык JavaScript. Создание динамических приложений-клиентов

Работа в интернете основана на схеме «клиент-сервер». Серверы – это программы, предоставляющие клиентам различные сервисы, например, сервер электронной почты, файловый сервер или веб-сервер. А клиенты – это потребители, программы или компьютеры, использующие то, что предлагают им серверы, например, различные браузеры.

Первоначально веб-страницы были статическими, всегда имеющими один и тот же вид.

Статический сайт - это сайт, который возвращает четко определенный неизменный контент с сервера всякий раз, когда запрашивается конкретный ресурс. Например, если есть интернет-магазин и страница с ноутбуком hp, то эта страница будет возвращена каждому пользователю, запросившему ее. Если понадобится добавить страницу с ноутбуком asus, то нужно будет создать новую

страницу и добавить ее сайт. Это не эффективно. Товаров тысячи, но на каждой странице товара есть сходная информация: о производителе, о цене, о наличии товара на складе, о способах доставки и т.д. Иными словами каждая страница товара версталась по одному и тому же шаблону. И если бы понадобилось изменить что-либо в структуре страницы — например, добавить новый раздел «связанные товары», то пришлось бы менять каждую страницу товара отдельно. Диаграмма архитектуры статического сайта представлена на рисунке 72.

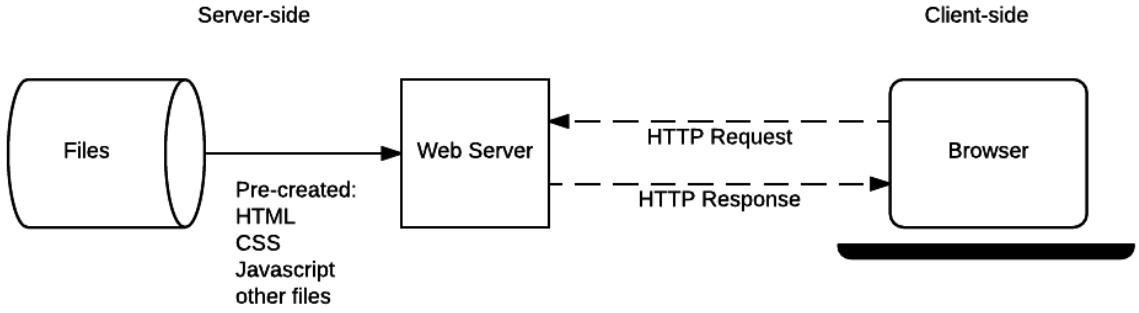


Рисунок 72 – Диаграмма архитектуры статического сайта

Фиксированные файлы хранятся на сервере, и по запросу браузера к серверу отправляются сервером в неизменном виде клиенту.

Затем с развитием популярности Интернета выросла и изменилась область применения веб-страниц.

Возникли динамические сайты. Динамический сайт — это сайт, который может генерировать и возвращать контент на основе конкретного URL-адреса запроса и данных, а не всегда возвращать один и тот же четко закрепленный код для определённого URL-адреса. Так, например, на сайте интернет магазина динамического сайта сервер хранит данные о товаре в базе данных. При получении GET-запроса для товара сервер определяет идентификатор товара, извлекает данные из базы данных и затем создаёт HTML-страницу для ответа, вставляя данные в HTML-шаблон. Это имеет большие преимущества перед статическим сайтом. Во-первых при работе с базой данных легко манипулировать с товаром, добавлять, удалять, искать и т.д. Во-вторых, использование HTML-шаблонов позволяет очень легко изменить структуру страницы, потому что это нужно делать только в одном месте, в одном шаблоне, а не через потенциально тысячи статических страниц.

На приведённой на рисунке 73 диаграмме показаны основные элементы веб-сайта и пронумерованны операции, которые происходят при формировании страницы клиента. В цепочку формирования страницы включаются веб-приложение (или программа на сервере, обрабатывающая HTTP-запросы и возвращающие HTTP-ответы), база данных, которая содержит информацию о товарах, и HTML-шаблоны страниц.

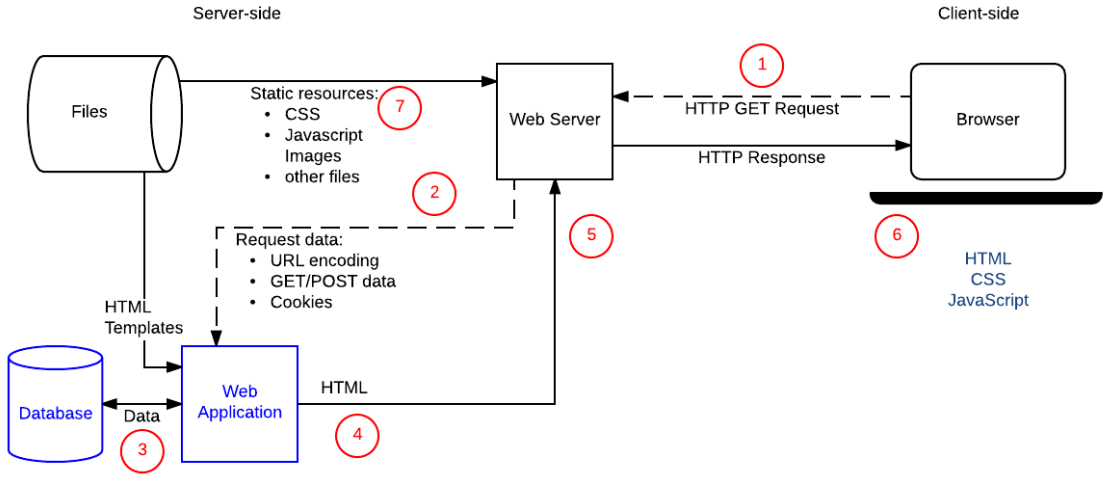


Рисунок 73. Основные элементы веб-сайта.

Допустим, пользователь интернет-магазина захотел увидеть все ноутбуки марки asus в определенном ценовом диапазоне. После того, как пользователь отправит форму с наименованием товара, маркой производителя и ценовым диапазоном, последовательность операций будет следующей:

1. Веб-браузер отправит HTTP-запрос GET на сервер с использованием базового URL-адреса ресурса и кодирования наименования товара, марки производителя и ценового диапазона в параметрах. Используется GET запрос, потому что речь идёт только о запросе выборки данных (а не об их изменении).
2. Веб-сервер определяет, что запрос является «динамическим» и пересылает его в веб-приложение для обработки.
3. Веб-приложение получает требуемую информацию из базы данных.
4. Веб-приложение динамически создаёт HTML-страницу, помещая данные из базы данных в заполнители внутри HTML-шаблона и возвращает сгенерированный HTML на сервер. Вместе со страницей возвращается код состояния HTTP 200 («успех»), или, какой-либо другой, если что-либо препятствует возврату HTML, например, «404».
5. Веб-сервер сгенерированную страницу отправляет в веб-браузер.
6. Веб-браузер обрабатывает возвращённый HTML, отправив отдельные запросы, чтобы получить любые другие файлы CSS или JavaScript, на которые он ссылается.

Активно стали применяться сценарии. Сценарии – это небольшие программы, записанные на каком-либо сценарном языке программирования, которые автоматизируют определенную задачу, которую без сценария пользователь делал бы вручную.

Сценарий имеют дело с программными компонентами, которые, однажды загруженные, в своей работе не зависят от дальнейшего подключения к Интернету. Скрипты создаются, передаются и выполняются, как простой текст. Для запуска скриптов не нужна компиляция или какая-либо другая специальная подготовка. В этом состоит основное отличие JavaScript от других языков программирования.

Первыми появились серверные сценарии. Они позволили получать и обрабатывать информацию от пользователей, что в свою очередь привело к тому, что появилась возможность создавать веб-страницы «на лету», в зависимости от запроса пользователя.

Серверные сценарии во многом расширили возможности веб-разработчиков. Но при этом значительно выросла нагрузка на веб-серверы, и увеличился сетевой трафик. Кроме того пользователи часто допускали ошибки или вводили неверную информацию, которая всё равно поступала на веб-сервер и обрабатывалась им. Все это привело к необходимости обрабатывать и проверять информацию на стороне клиента до её отправки на сервер.

JavaScript был создан в 1995 году программистами компании Netscape. Сам язык изобрел Брендон Эйх (Brendan Eich). Помимо Брендон Эйха в разработке участвовали сооснователь Netscape Communications Марк Андрисин и сооснователь Sun Microsystems Билл Джой. Перед ними стояла цель: за 10 дней создать «язык для склеивания» составляющих частей веб-ресурса (изображений, плагинов, Java-апплетов), который был бы удобен для веб-дизайнеров и программистов, не обладающих высокой квалификацией. Первоначально язык назывался LiveScript и предназначался как для программирования на стороне клиента, так и на стороне сервера. На синтаксис языка оказали влияние языки C и Java, и, поскольку Java в то время было модным словом, 4 декабря 1995 года LiveScript переименовали в JavaScript, получив соответствующую лицензию у Sun.

Впервые новый язык был использован в браузере Netscape Navigator 2.0. Это дало браузеру Netscape Navigator дополнительные преимущества на рынке сбыта. Из-за этого разработчики Internet Explorer включили в него поддержку другого, но по большей части совместимого с JavaScript языка. Поскольку JavaScript имел лицензию Sun, компания Microsoft назвала свою разработку JScript. После этого JScript стал использоваться во всех последующих браузерах от Microsoft, начиная с Internet Explorer 3.0.

Далее в погоне за прибылью в каждой из последующих версий Netscape и Internet Explorer стали появляться новые возможности, несовместимые с браузером-конкурентом. Делалось это для того, чтобы заставить разработчиков включать в свои страницы новые технологии, доступные только в одном из браузеров, поддерживающем язык JavaScript или JScript. Фактически, компании выкидывали и продолжают выкидывать деньги на то, чтоб создать несовместимый код. Чтобы обеспечить совместимость версий языка независимых разработчиков, Генеральной Ассамблеей ECMA (Ecma International – ассоциация, деятельность которой посвящена стандартизации информационных и коммуникационных технологий) был создан стандарт. Стандартизированная версия имеет название ECMAScript, описывается стандартом ECMA-262. Она основана на нескольких базовых технологиях, наиболее известными из которых являются JavaScript (Netscape) и JScript (Microsoft). Стандартизированная версия JavaScript, называемая ECMAScript, работает одинаково во всех приложениях, поддерживающих стандарт.

Современный JavaScript может выполняться не только в браузере или на сервере, но фактически на любом устройстве, которое имеет специальную программу, называющуюся «движком» JavaScript. Движок JavaScript браузера иногда называют «виртуальной машиной JavaScript». В браузере Mozilla Firefox движок JavaScript имеет имя SpiderMonkey. Это первый в истории движок JavaScript. В Google Chrome, Opera и Edge движок JavaScript называется V8, а в Safari, например, SquirrelFish.

При запуске JavaScript-программы в браузере встроенный в браузер движок JavaScript читает текст скрипта и преобразует его в машинный язык, а затем запускает машинный код. При работе движок сам просматривает код и оптимизирует его. Скрипты работают быстро.

Как уже говорилось, скрипты (сценарии) написанные на JavaScript не компилируются в исполняемую программу, а хранятся в текстовых файлах с расширением .js или в виде вставленных фрагментов в html-код и передаются движку каждый раз при запуске сценария на исполнение. Программа на JavaScript не имеет классической стартовой функции main присущей любому компилируемому языку, с которой начинается работа программы.

JavaScript включает стандартную библиотеку объектов, например, Array, Date и Math, а также базовый набор операторов и управляющих конструкции. Ядро JavaScript может быть расширено для различных целей путём добавления в него новых объектов. JavaScript на стороне клиента расширяет ядро языка, предоставляя объекты для контроля браузера и его Document Object Model (DOM). Клиентские расширения позволяют приложению, например, размещать элементы в html-форме и обрабатывать пользовательские события, такие как щелчок мыши, ввод данных в форму и навигация по страницам и др. Расширение javaScript на стороне сервера позволяет приложению, например, соединяться с базой данных, обеспечивать непрерывность информации между вызовами приложения или выполнять манипуляции над файлами на сервере и др.

1.4.1. Источники информации по возможностям языка

Основным источник информации о JavaScript является Спецификация ECMA-262 https://www.ecma-international.org/publications-and-standards/standards/ecma-262/.

Справочник по JavaScript с примерами MDN (Mozilla) JavaScript Reference https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference.

Учебник по JavaScript с примерами https://learn.javascript.ru/

1.4.2. Возможности JavaScript в браузере

JavaScript называют безопасным языком программирования, поскольку он не предоставляет низкоуровневый доступ к памяти или процессору, т.к. изначально был создан для браузеров, не требующих этого.

С помощью JavaScript в браузере доступно всё, что связано с манипулированием веб-страницами, взаимодействием с пользователем и веб-сервером, например, можно:

добавлять новый html-код на страницу;
изменять контент страницы;
изменять стили на странице;
реагировать на действия пользователя (щелчки мыши, перемещения указателя, нажатия клавиш и т.д.);
отправлять запросы на сервер;
скачивать и загружать файлы;
получать и устанавливать куки;
задавать вопросы пользователю;
показывать сообщения;
запоминать данные на стороне клиента.

1.4.3. Ограничения javaScript в браузере

Возможности javaScript в браузере были сознательно ограничены в целях безопасности, а именно в предотвращении доступа недобросовестной веб-страницы к личной информации или нанесения ущерба данным пользователя:

- javaScript на веб-странице не может читать и записывать произвольные файлы на жёстком диске, копировать их или запускать программы;
- javaScript не имеет прямого доступа к системным функциям операционной системы;
- работа с файлами предполагает ограниченный доступ: только при совершении пользователем определенных действий, таких как перетаскивание файла в окно браузера или выбор файла помощью тега <input>;
- при организации взаимодействия с дополнительными устройствами, например, такими камера или микрофон, требуется явное разрешение пользователя, что исключает возможность незаметно включать эти устройства;
- javaScript с одной страницы в браузере не имеет доступа к другой странице, открытой в другой вкладке;
- в случае, если при помощи javaScript, открывается другая страница, то javasScript с открывающей страницы не имеет доступа к открытой им странице, если страницы принадлежат разным сайтам;
- javaScript может отправлять и получать данные с сервера с которого пришла текущая страница, но его способность получать данные с других сайтов ограничена, поскольку для этого требуется явное согласие, выраженное в заголовках HTTP.

У современных браузеров есть расширения, с помощью которых можно запрашивать дополнительные разрешения для действий, на которые наложены ограничения.

Эти ограничения на javaScript действуют только в браузерах. Вне браузеров, например, на сервере, таких ограничений нет.

Сегодня JavaScript является самым распространённым языком программирования, используемым в браузерах. Он полностью приспособлен для взаимодействия с HTML и CSS.

Для удобства разработчиков, работающих на других языках программирования, но которым нужно использовать JavaScript, были разработаны и продолжают создаваться языки, которые позволяют конвертировать код на другом языке в JavaScript «под капотом». Например, Brython. Он конвертирует код на Python в JavaScript, что позволяет писать приложения на чистом Python без JavaScript.

1.4.4. Редакторы кода

Для набора текстов программ JavaScript можно использовать редакторы кода. Редакторы могут быть различными по степени сложности. Если предполагается использовать JavaScript в большом проекте, то можно использовать мощные редакторы или интегрированные среды разработки или IDE (Integrated Development Environment). IDE позволяет работать с проектом целиком, как правило, имеет автодополнение по коду, интегрирована с системой контроля версий (например, с git), средой для тестирования и другими инструментами на уровне всего проекта. Примером бесплатной IDE является Visual Studio Code, платной - WebStorm. Если требуется написание сравнительно небольших программ, то можно использовать небольшие простые редакторы, например, Notepad++.

1.4.5. Консоль JavaScript в браузерах

По умолчанию все ошибки, допущенные в коде, браузер пытается решить сам. Он интерпретирует код тем или иным образом, и исправляет код по собственным алгоритмам, либо игнорирует неверные инструкции, как в случае с CSS. Сами ошибки в браузере не видны. О наличии ошибок разработчик может догадаться только по тому, что страница отображается неверно, не работает какая-то функциональность. А это значит, что, если что-то будет не работать или работать не так было задумано, невозможно явно увидеть причину, из-за которой возникла проблема. В браузер встроены ***Инструменты разработчика (Developer tools)***, которые позволяют работать с кодом страницы, смотреть ошибки, выполнять различные команды, изменять код и многое другое. В операционной системе Windows в браузерах Chrome и Firefox открыть Инструменты разработчика можно открыть нажав клавишу F12 или Cmd+Opt+J, если используется Mac. Работать с JavaScript можно через консоль разработчика. Для этого надо перейти на вкладку ***Консоль***. Эта вкладка открывается по умолчанию. Консоль разработчика отображает информацию о загруженной веб-странице.

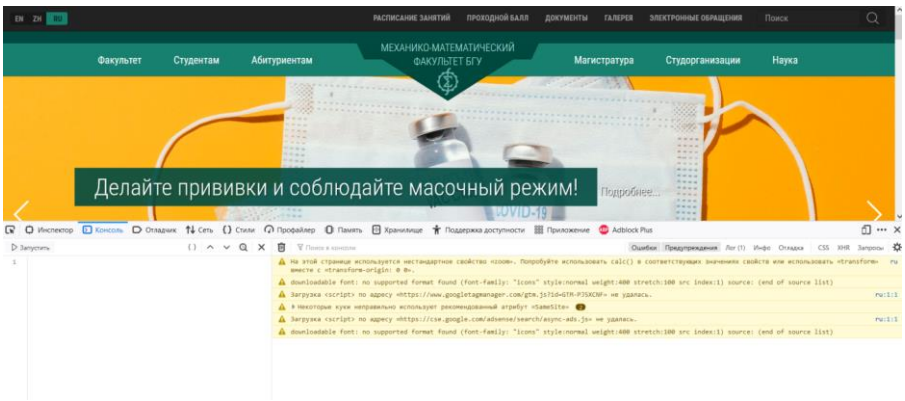


Рисунок 73

Как видно на рисунке 73, консоль инструменты разработчика отображают в том числе и ошибки, допущенные в коде JavaScript. Кроме этого Консоль разработчика включает командную строку, которую можно использовать, чтобы выполнить код JavaScript.

Как использовать панель консоли при отладке:
https://habr.com/ru/company/ruvds/blog/486692/
https://habr.com/ru/company/otus/blog/520706/

1.4.6. Размещение JavaScript кода на веб-странице

Время и условия выполнения скрипта зависит от того, когда и как этот интерпретатор JavaScript получает управление. Что, в свою очередь, зависит от размещения кода. В общем случае можно выделить четыре способа размещения JavaScript-кода:

в гипертекстовых ссылках (схема url);
в обработчиках событий (в атрибутах событий);
в теге <script>;
внешний JavaScript.

1.4.7. Url-схема "javascript:"

При обнаружении браузером на веб-странице ссылки вызывается стандартная программа реализации гипертекстового перехода. JavaScript позволяет поменять стандартную программу на программу пользователя. Для этого в атрибуте href надо указать javaScript-код следующим образом: `<a href="javascript: код_программы">...</a>`, где `код_программы` – это программа JavaScript, который исполняется при выборе гипертекстовой ссылки.

При нажатии на гипертекстовую ссылку «Это важно» появляется окно предупреждения «Внимание!».

```
<a href="javascript:alert('Внимание! Не забудьте сделать домашнее задание.')">Это важно</a>
```

Такую схему можно использовать и в некоторых других атрибутах, например, в атрибуте ***action***:

```
<form name="mainForm" action="javascript:
   if (document.getElementById('textFieldId').value=='')
    alert('Вы забыли заполнить поле!'); void(0);">
```

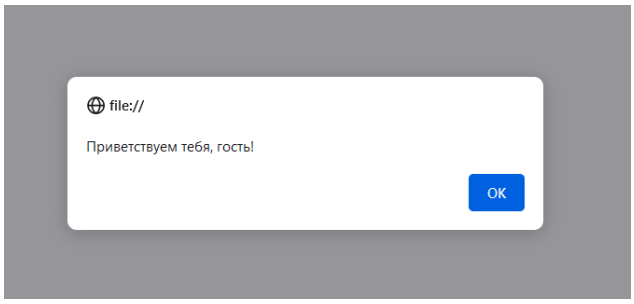
```
  <input type="text" id="textFieldId" size="30">
  <p>
    <input type="submit" value="Заполнить">
    <input type="reset" value="Очистить">
  </p>
</form>
```

1.4.8. JavaScript в обработчиках событий

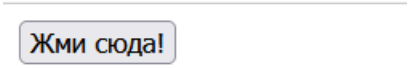
JavaScript- функции могут размещаться в обработчиках событий. Например, в момент завершения полной загрузки документа происходит событие Load и вызывается обработчик события onLoad, при нажатии на кнопку происходит событие Click и соответственно вызывается обработчик этого события onClick и т.п.

```
<body onLoad="alert('Приветствуем тебя, гость!');">
  <input type="button" value="Жми сюда!"
    onClick="alert('Вы нажали кнопку');">
</body>
```

Первый экран.



Второй экран.



1.4.9. Контейнеры <script>

Программы на JavaScript могут внедряться в html-документ при помощи тега <script>. Теги <script> могут быть размещены в любом месте внутри тега <body> или внутри тега <head> и использоваться любое количество раз.

Если при разборе документа html-парсер встречает тег <script>, он передаёт управление JavaScript-интерпретатору. После этого JavaScript-интерпретатор выполняет код внутри контейнера <script> и возвращает управление html-парсеру.

1.4.10. Внешний JavaScript

Сценарии стоит размещать во внешних файлах. Это практично, поскольку один и тот же сценарий можно использовать на разных веб-страницах. Присоединяются внешние файлы со скриптами при помощи тега <script>. Адрес заносится в атрибут src. Если присутствует атрибут src, то содержимое контейнера

<script> должно быть пустым, поскольку согласно спецификации HTML, если скрипт подключается из внешнего файла, то все, что написано между тэгами <script> и </script> будет проигнорировано браузером. Внешний скрипт можно присоединять внутри элементов <head> или <body>. Можно присоединять несколько файлов со javascript-кодом на страницу.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <script>
      document.writeln("Выполнен скрипт из head");
    </script>
    <script src="Script1.js"></script>
  </head>
  <body>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML =
        "Выполнен скрипт из body";
      </script>
    <noscript>Извините, ваш браузер не поддерживает
      JavaScript! Отображение страницы будет
      не полным.
    </noscript>
    <script src="Script2.js"></script>
    <script src="Script3.js"></script>
  </body>
</html>
```

Файлы сценариев:

```
// JavaScript Document Script1.js
document.writeln("Выполнен скрипт из Script1.js подключенный в
head");
// JavaScript Document Script2.js
document.writeln("Выполнен скрипт из Script2.js подключенный в
body");
// JavaScript Document Script3.js
document.writeln("Выполнен скрипт из Script3.js подключенный в
body");
```

Преимущества внешнего JavaScript

Размещение сценариев JavaScript во внешних файлах имеет ряд преимуществ:

это разделяет HTML и JavaScript;

воспринимать такой код проще;

это удобно при поддержке кода;

загрузка последующих веб-страниц, куда подключен javascript-код происходит быстрее, поскольку файлы сценариев уже есть в кэше.

1.4.11. Способы ввода информации и вывода результатов

Используя JavaScript результат выполнения кода можно вывести:

на страницу при помощи innerHTML;

на страницу при помощи document.write();

в окно сообщений, например. окно предупреждения при помощи window.alert();

в консоль разработчика при помощи console.log() или console.table().

Следующий код выводит результат работы скрипта на страницу при помощи innerHTML.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <script>
      function dateDemo(){
        let today = new Date();
        let dd = String(today.getDate());
        let mm = String(today.getMonth() + 1);
        let yyyy = today.getFullYear();
        today = mm + '/' + dd + '/' + yyyy;
        return today;}
    </script>
  </head>
  <body>
    <p id="demo">Этот текст пропадет, а выведется дата</p>
    <script>
  document.getElementById("demo").innerHTML ='1. ' + dateDemo()
    </script>
    <script>
      alert('3. ' + dateDemo());
    </script>
    <script>
      console.log('4. ' + dateDemo());
    </script>

  </body>
</html>
```

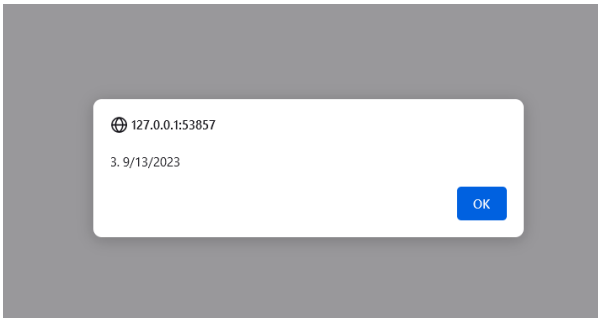
В свойстве innerHTML хранится в виде строки содержимое элемента, к которому применяется свойство. Свойство доступно для чтения и записи, поэтому можно получать и изменять содержимое элемента. Метод выводит на страницу переданные ему аргументы. На этапе загрузки страницы метод добавляет контент.

Объект window является глобальным в браузерах. Но к нему нельзя обратиться напрямую. Обратиться к нему можно при помощи его свойства window, которое ссылается на сам объект. Метод alert() объекта window выводит модальное диалоговое окно с сообщением и кнопкой Ok.

Объект console служит для доступа к средствам отладки браузера. Доступ к console можно получить через свойство глобального объекта window:

window.console или просто console. Console.log выводит сообщение в веб-консоль.

Первый экран.



Второй экран.

1. 9/13/2023

2. 9/13/2023

1.4.12. Системные диалоговые окна

Окна сообщений позволяют выводить различные сообщения, в том числе и результаты работы, а также при необходимости вводить данные. Диалоговые окна бывают трех типов:

- окно с сообщением и кнопкой Ok;
- окно с сообщением и кнопками Ok и Cancel;
- окно с сообщением, полем ввода и кнопками Ok и Cancel.

Эти диалоговые окна выводятся при помощи методов alert, confirm и prompt соответственно. Каждое из них является синхронным и модальным, то есть загрузка дальнейшей части страницы приостанавливается на время показа такого окна и возобновляется после его закрытия. Кроме этого, они ещё приостанавливают загрузку дальнейшей части страницы.

Дизайн этих диалоговых окон очень простой и непритязательный. Внешний вид окна зависит от браузера, и разработчик не может изменить его вид и переделать дизайн в стиле сайта. Это не всегда допустимо, но это самый простой способ вывести сообщение или получить информацию от пользователя, поэтому их используют в тех случаях, когда дизайн не особо важен. Место, куда будет выведено окно сообщений также выбирает браузер.

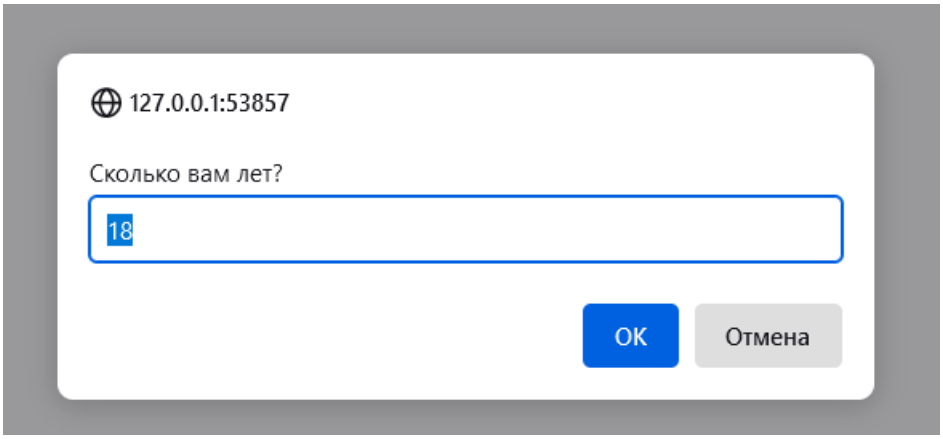
Метод prompt() выводит стандартное диалоговое окно, с сообщением и кнопками Ok и Cancel и требует от пользователя ввод текста. Как и два предыдущих метода, метод prompt() является методом класса window, причем имя класса почти всегда можно опускать. Синтаксис метода имеет вид:

prompt(message, default);

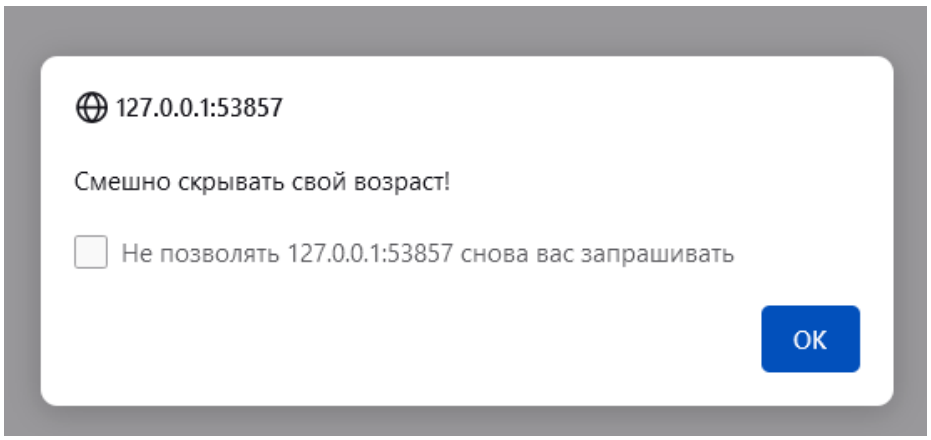
где message – строка, которая будет выведена в качестве сообщения, а необязательный параметр default – это строка, число или значение свойства, определяющее значение поля ввода по умолчанию. Если параметр default не определен, то диалоговое окно отображает значение типа undefined (поле пустое). Ме-

тод возвращает введенное значение или null, если пользователь отказался от ввода и нажал на кнопку Cancel.

```
<script>
  let years=prompt('Сколько вам лет?','18');
  if (years == ''){
    alert('Так много лет, что затрудняетесь посчитать?
    Или так мало, что плохо считаете?');}
  else if(years == null){
    alert('Смешно скрывать свой возраст!');}
  else {alert('Ого! Вам '+years+'! Поздравляем!');}
</script>
```



Выбираем отмена.



1.4.13. Основные синтаксические правила

Инструкции – это синтаксические конструкции и команды, которые выполняют действия. Код javascript может содержать множество инструкций, каждая инструкция завершается точкой с запятой. Чтобы код было легче читать, каждую инструкцию пишут на отдельной строке. В этом случае точку с запятой можно не ставить, поскольку браузеры могут распознать переход на следующую строку как неявную точку с запятой. Но в JavaScript есть возможность переносить часть слишком длинных инструкций на следующую строку (обычно это делается после знака операции). В случае такого разрыва инструкции JavaScript может ошибочно поставить точку с запятой. И код будет интерпретироваться не так, как вам бы того хотелось, или вовсе станет ошибочным. В следующем примере JavaScript добавляет точку с запятой после return. А при отсутствии выражения после return инструкция возвращает значение undefined.

Поэтому синтаксической ошибки нет, но функция будет работать не так, как ожидалось.

```
<script>
  function f(x){
      return
    x+0.1;
  }
  document.write('новое значение x='+x+'<br>');
</script>
```

Поэтому, несмотря на то, что заключение операторов точкой с запятой не требуется, а только настоятельно рекомендуется, все же стоит ставить точки запятой после каждого оператора.

JavaScript, как и многие языки, в некоторых случаях игнорирует дополнительные пробелы в инструкциях. Поэтому желательно добавлять пробелы в код, чтобы сделать его более читабельным. Хорошая практика – ставить пробелы вокруг операторов (= + – * /):

```
let y = 1; // let y=1; воспринимается хуже
let x = y + z; // let x=y+z; воспринимается хуже
```

1.4.14. Комментарии

Комментарии в JavaScript Си-подобные. Для обозначения однострочного комментария используется двойной слеш. Для выделения многострочного комментария используется /* и */. Вложенные комментарии не поддерживаются.

```
<script>
/*
  /* Ошибка! Вложенный комментарий не допускается. */
*/
  x1 = 1; //Первое число Фибоначчи
  x2 = 1; /*Второе число Фибоначчи*/
  document.write(x1 + "  ");
  document.write(x2 + "  ");
  /*Начиная с третьего,
    находим числа Фибоначчи в цикле, */
  for(i = 3; i < 11 ; i++){
    x3 = x1 + x2;
    document.write(x3 + "  ");
    x1 = x2;
    x2 = x3;
  }
</script>
```

1.4.15. Зарезервированные слова

Зарезервированные слова не могут быть именами переменных, функций и меток циклов. Зарезервированные слова JavaScript: ***abstract***, ***arguments***, ***await***, ***boolean***, ***break***, ***byte***, ***case***, ***catch***, ***char***, ***class***, ***const***, ***continue***, ***debugger***, ***default***, ***delete***, ***do***, ***double***, ***else***, ***enum***, ***eval***, ***export***, ***extends***, ***false***, ***final***, ***finally***, ***float***, ***for***, ***function***, ***goto***, ***if***, ***implements***, ***import***, ***in***, ***instanceof***, ***int***, ***interface***, ***let***, ***long***, ***native***, ***new***, ***null***, ***package***, ***private***, ***protected***, ***public***, ***return***, ***short***, ***static***, ***super***, ***switch***,

synchronized, ***this***, ***throw***, ***throws***, ***transient***, ***true***, ***try***, ***typeof***, ***var***, ***void***, ***volatile***, ***while***, ***with***, ***yield***.

Кроме того, следует избегать использования идентификаторов встроенных объектов, свойств и методов, предопределенных в языке JavaScript: ***Array***, ***Date***, ***eval***, ***function***, ***hasOwnProperty***, ***Infinity***, ***isFinite***, ***isNaN***, ***isPrototypeOf***, ***length***, ***Math***, NaN, ***name***, ***Number***, ***Object***, ***prototype***, ***String***, ***toString***, ***undefined***, ***valueOf***. Если попытаться создать переменную или функцию с таким именем, то это будет приводить либо к ошибке (если свойство определено как доступное только для чтения), либо к переопределению.

Не стоит использовать как идентификаторы также имена объектов и свойств HTML, объекта Window и имена обработчиков событий HTML. Не желательно использовать ключевые слова из языков программирования, поскольку JavaScript можно использовать как язык программирования во многих приложениях.

1.4.1. Идентификаторы

На самом деле переменных как таковых в js НЕТ, есть идентификаторы. Но для простоты в этот момент мы не будем углубляться. Имена переменных могут содержать буквы, цифры, знаки подчеркивания и доллара. Цифра не может быть первым символом. В качестве имен нельзя использовать зарезервированные слова. В JavaScript используется набор символов Unicode, поэтому при написании кода их можно использовать.В идентификаторах также допустимо использовать символы Unicode, например, å или ü. Например, слово Gemüse означает по-немецки овощ. Его можно использовать в качестве имени переменной.

```
var Gemüse = "Kartoffel";
```

По этой же причине можно использовать русские или белорусские буквы.

```
var овощ = "картофель";
var воўк  = "Серы";
```

Переменные в JavaScript могут быть объявлены тремя способами:

Первый способ: с использованием ключевого слова ***let*** или ***const***. Например, `let userLogin.` Данный синтаксис может быть использован для объявления переменной в области видимости блока. До строки с объявлением переменной интерпретатор JavaScript создаёт TDZ (Temporary Dead Zone – часть в которой будет проверяться не вызывает ли программист какую-то функцию в которой используется эта переменная до объявления)

Второй способ: объявление вообще без указания ключевых слов. Например, `x = 42.` Переменные, объявленные данным способом, являются глобальными. Такое объявление генерирует строгое предупреждение, поэтому его не рекомендуется использовать.

Третий способ: с использованием ключевого слова ***var***. Например, `var userName.` Такой синтаксис используется для объявления как локальных, так и глобальных переменных.

При объявлении можно сразу присвоить переменной значение, совместить объявление с инициализацией.

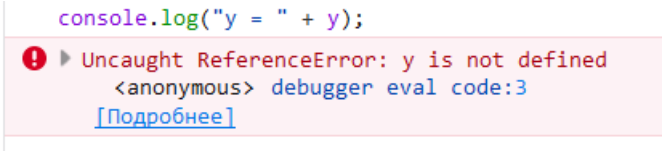
```
let userLogin = "guest";
```

Если при объявлении переменной ей не было присвоено никакого значения, она будет иметь специальное значение ***undefined***, до тех пор, пока ей не будет присвоено другое значение:

```
var a;
let b;
console.log(a);   // undefined
console.log(b);   // undefined
```

При попытке доступа к необъявленной переменной или переменной , определенной с помощью, ***const*** или ***let*** до её объявления будет выброшено исключение ***ReferenceError***.

```
console.log("y = " + y);
```



Переменные, определенные с помощью, ***let*** должны быть объявлены перед использованием. Следующий код вызовет ошибку ***ReferenceError*** при попытке использовать переменную, объявленную в коде ниже при помощи ключевого слова ***let***.

```
console.log("The value of x is " + x); //Uncaught ReferenceError: x
не определена
let x;
```

В JavaScript можно использовать переменные, которые будут объявлены в коде ниже при помощи ключевого слова var. Это называется поднятием или *всплытием переменных*. Переменные, которые ещё не были инициализированы, возвратят значение undefined:

```
console.log("z = " + z);
var z = 20;
```

Переменные в JavaScript не имеют типа, поэтому переменной может быть присвоено значение любого типа, а затем этой же переменной может быть присвоено значение уже другого типа:

```
var x = 1;
let y = 1.1;
x = 5.6;
y = "content";
```

Языки, в которых типы данных существуют, но переменные не привязаны ни к одному из типов данных, называются «динамически типизированными». JavaScript – динамически типизированный язык программирования.

В реальном проекте большая часть времени тратится на сопровождение кода, т.е. на его изменение и расширение, а не на написание нового кода с нуля. Причем ищут первоначальный код одни люди, сопровождают его – совершенно другие. Когда просматривается чужой код, или свой, но после какого-то промежутка времени, гораздо легче понять его, если он хорошо размечен. Этому способствуют хорошие имена переменных.

Перечислим основные правила хорошего наименования.

- Имена должны легко читаться, например, `userName`.
- Избегайте использования аббревиатур или коротких имён, таких как a, b, c.
- Если идентификатор содержит несколько слов, обычно используется верблюжья нотация, то есть, слова следуют одно за другим, где каждое следующее слово начинается с заглавной буквы: `warningSigns`.
- Имена должны быть максимально описательными и лаконичными. Например, не стоит использовать `data` для обозначения даты рождения или `value` для обозначения затрат. Такие имена ничего не говорят. Лучше использовать `birthday` или `amountCosts`.

Именование переменных – это один из самых важных и сложных навыков в программировании. Быстрый взгляд на имена переменных может показать, какой код был написан новичком, а какой – опытным разработчиком.

1.4.2. Константы

При объявлении констант используется слово const.

```
const MY_PET = 'puppy';
```

Константы используются в качестве псевдонимов значений, которые известны до начала исполнения скрипта. Это позволяет избежать использования «магических чисел». Названия констант пишутся с использованием заглавных букв и подчёркивания. Например, для обозначения призывного возраста подойдёт константа MILITARY_AGE.

```
const MILITARY_AGE = 18;
```

Названия таких констант пишутся с использованием заглавных букв и подчёркивания.

Также в программах встречаются переменные, которые получают вычисляемое значение один раз и затем не изменяются. Их называют константными или неизменяемыми переменными. Константные переменные объявляются при помощи ключевого слова const, но именовать их принято по правилу именования переменных, т.е. с использованием верблюжьей нотации.

```
const numberAwards = 3; //количество наград
```

1.4.3. Типы данных

Всего существует 8 типов данных (точнее читаем в разделе 4.3 и 6.1- A primitive value is a member of one of the following built-in types: **Undefined**, **Null**, **Boolean**, **Number**, **BigInt, String**, and **Symbol**):

- `number` для целых и вещественных чисел,
- `bigint` для работы с целыми числами произвольной длины,
- `string` для строк,
- `boolean` для логических значений истинности или ложности: `true/false`,
- `null` – тип с единственным значением `null`, т.е. «пустое значение» или «значение не существует»,

- `undefined` – тип с единственным значением `undefined`, т.е. «значение не задано»,
- `symbol` – уникальные идентификаторы; их мы не будем изучать.
- `object` – всё остальное

Оператор `typeof` возвращает тип значения переменной, с двумя исключениями:

```
 typeof null == "object" // "ошибка" в языке, точнее фича потому,что
null это действительно "пустой" объект

 typeof (()=>{}) ------> "function" // сделано именно для функций
typeof function fff(){} ------> "function" // сделано именно для
функций
```

Тип данных переменной зависит от значений, которые она принимает. Тип переменной может изменяться в ходе выполнения программы. Преобразование типов выполняется автоматически. И зависит от контекста в котором используется переменная. Например, при суммировании числовых и строковых значений числовые значения преобразуются в строковые:

```
let message = 10 + " дней до отпуска";
typeof (10 + " дней до отпуска"); // вернет "string"
```

В ходе выполнения операций происходит динамическое приведение типов. Но следует помнить, что JavaScript оценивает выражения слева направо, производя приведение типов последовательно.

```
1 + 38 + ' попугаев'; // 39 попугаев
'1' + 38 + ' попугаев'; // 138 попугаев
```

В случае `1 + 38 + ' попугаев'` JavaScript обрабатывает 1 и 38 как числа, пока не достигнет « попугаев». В случае `'1' + 38 + ' попугаев'` – первый операнд является строкой, поэтому все операнды обрабатываются как строки.

Получить тип данных переменной или выражения можно с помощью оператора typeof.

Идентификатор переменной в JavaScript пости всегда ссылается на некую область в памяти в которой может находиться или примитивные данные или опять ссылка на «составной» тип данных object. К простым типам данных относят строковый, числовой, логический, symbol и значения null и undefined.

Разница в работе с примитивными и составными типами видна при копировании значений. Когда переменной присваивается значение примитивного типа, например число, то в переменную записывается ссылка на само значение. Именно поэтому понятно почему строки в Js являются иммутабельными, поскольку в переменной содержится только ссылка на область константной памяти в которой находится сама строка. При выполнении присваивания одной переменной со значением простого типа другой переменной происходит копирование значения. Для строк – копируется фактически только указатель на строку в памяти, а не сама строка. В результате каждая числовая переменная имеет свою копию значения и изменения в одной из переменных никак не сказывается на значении другой:

```
let amountFirst = 100;
let amountSecond = amountFirst; // Копируем значение
document.write("У вас на счету: " + amountFirst + // 100
     "<br> Вы можете потратить: " + amountSecond); // 100
amountFirst = 20; // Изменяем значение
document.write("<br><br>У вас на счету:"+amountFirst+//20
      "<br> Вы можете потратить: " + amountSecond); // 100
```

Когда переменной присваивается значение типа object, то в переменную записывается ссылка на значение. При выполнении присваивания одной переменной копируется ссылка, а не окончательные примитивные значения всех полей. В результате обе переменные ссылаются на одно и то же место в памяти и любые изменения *полей* одной из переменных автоматически изменят вторую переменную. Но если изменение затрагивает не *поля*, а происходит присваивание *самого* второго объекта, то это *не влияет* на первый, поскольку второй объект будет теперь ссылаться на новую область памяти.

```
let ob1 = {x:0,y:0};
let ob2 = ob1; // Копируем ссылку на объект
console.log(ob1)//(0,0)
console.log(ob2)//(0,0)
ob1.x = 10; //Изменяем значение первой координаты
console.log(ob1)//(10,0) Изменился и ob1
console.log(ob2)//(10,0) Изменился и ob2
ob2.y = 20;// Изменяем значение второй координаты ob2
console.log(ob1)//(10,20) Изменился и ob1
console.log(ob2)//(10,20) Изменился и ob2
ob2 = {x:30,y:30};// теперь ob2 ссылается на совершнггл новое место
в памяти никак не связанное с ob1
console.log(ob1)//(10,20) Изменился и ob1
console.log(ob2)//(30,30) Изменился и ob2
```

1.4.4. Числовой тип (Number)

В JavaScript тип number не может содержать числа больше, чем 2^{53}-1 или меньше, чем -2^{53}-1. Это ограничение вызвано их внутренним представлением. С изменение разрядности операционной системы диапазон изменится, но ограничения останутся.

В JavaScript к числовому типу number относятся целые числа и числа с плавающей точкой.

```
typeof 12345 // "number"
typeof -12.345 // "number"
typeof 12345e-3 // "number"
typeof 07 // "number"
typeof 0xa // "number"
```

Если в записи числа содержаться только десятичные цифры, и оно не начинается с префикса 0 или 0x, то это десятичное целое число. Целые числа могут быть положительными, отрицательными или нулем. Десятичные числа с плавающей точкой содержат в качестве разделителя точку или записываются в научной нотации. Буква e в экспоненциальном формате может быть прописной или строчной.

Шестнадцатеричные целые числа начинаются с префикса 0x (ноль+икс), причем буква X может быть как большой, так и малой. В запись шестнадцатеричного числа могут входить цифры от 0 до 9 и буквы от A до F, либо строчные, либо прописные. Поскольку буква e обозначает цифру в шестнадцатеричной записи, то шестнадцатеричные числа не могут быть представлены в экспоненциальном формате. Шестнадцатеричные числа могут быть как положительными, так и отрицательными.

Восьмеричные целые числа начинаются с нуля и могут содержать цифры от 0 до 7. Если первая цифра числа нуль, но в записи числа присутствуют цифры 8 или 9, то считается, что число записано в десятичной системе счисления с незначащим нулем впереди. Восьмеричные числа могут быть отрицательными, но не могут быть числами с плавающей точкой или быть записанными в экспоненциальном формате.

В JavaScript определены также специальные *числовые значения*:

NaN, или неверное знасение числа (Not a Number);

Infinity (положительная бесконечность);

-**Infinity** (отрицательная бесконечность);

0 (положительный 0);

-0 (отрицательный 0).

В отличие от других языков программирования математические операции в JavaScript *не могут привести к ошибке или аварийно завершить работу программы*. Если математическая операция не может быть совершена, то возвращается специальное значение NaN, которое обозначает математическую ошибку. Например, деление 0/0 в математическом смысле не определено, поэтому результатом операции будет NaN, тоже касается и Infinity/Infinity. Значение NaN не равно ничему, включая себя. Значение NaN можно проверить только специальной функцией isNaN(), которая возвращает true если аргумент NaN, и false для любого другого значения. Результат любой операции с NaN также будет равен NaN. Значение NaN можно получить также как Number.NaN.

```
typeof (1/0) // NaN
0/0 // NaN
isNaN(0/0) // true
Number.NaN // NaN
NaN+1 // NaN
NaN == NaN // false
NaN === NaN // false
```

В теоретических математических выкладках при делении получается бесконечность. Аналогично в JavaScript значения выражений, в которых происходит деление на нуль, также равно бесконечности со знаком плюс или минус. Infinity – особенное численное значение, которое ведет себя в точности как математическая бесконечность. Когда вещественное число превышает самое большое представимое конечное значение, результату присваивается специальное значение бесконечности, которое в JavaScript обозначается как Infinity. А когда отрицательное число становится меньше наименьшего представимого отрицательного числа, результатом является отрицательная бесконечность -

Infinity. Infinity больше любого числа. Добавление к бесконечности любого числа не меняет ее. Бесконечность можно присвоить в явном виде:

```
let x = Infinity;
```

Также специальные значения, обозначающие плюс и минус бесконечность, можно получить как константы Number.POSITIVE_INFINITY и Number.NEGATIVE_INFINITY.

Значение Infinity / Infinity не определено. Сумма бесконечностей одного знака есть бесконечность того же знака. Разность бесконечностей противоположного знака не определена.

```
1/0 // Infinity
-1/0 // -Infinity
Infinity > 1e100 // true
Infinity + 5 // Infinity
Infinity – 1e100 // Infinity
Infinity / Infinity // NaN
Infinity + Infinity // Infinity
Infinity – Infinity // NaN
Infinity * Infinity; // Infinity
```

Для чисел с плавающей запятой в JavaScript определены два нуля: положительный и отрицательный. Но, как правило, в программе это не нужно учитывать. Оба нуля при сравнении равны и на выводе они тоже выглядят одинаково. Знак нуля важен только для операций деления и умножения. Можно определить константу со значением −0, и ее знак будет учитываться при делении.

```
1/ Infinity // 0
1/ -Infinity // 0
1/ Infinity == -1// false
1 / 0 // Infinity
1 / (-0) // -Infinity
```

1.4.5. BigInt

BigInt позволяет работать с целыми числами произвольной длины.

Чтобы создать значение типа BigInt, необходимо добавить n в конец числового литерала:

```
typeof 1n // "bigint"
```

Данные типа BigInt можно создать также вызвав функцию BigInt:

```
BigInt("12345678912345678");// 12345678912345678n
```

В математических операциях нельзя смешивать BigInt и обычные числа. Операция деления возвращает округлённый результат, без дробной части. Операции сравнения < и > работают так как нужно. Но равенство с обычными типами достигается только при нестрогом сравнении == без приведения типов.

```
1 == 1n  // true
1 === 1n // false
```

1.4.6. Строковый тип

Имя типа string. Строки – это произвольные текстовые данные заключенные в кавычки. Кавычки могут быть одинарными или двойными.

Отдельно символьного типа в JavaScript нет. Один символ также является строкой. Все строки, вне зависимости от кодировки, хранятся в формате юникод. В строки могут включаться специальные символы:

\b – Backspace или возврат курсора на одну позицию;

\f – Form feed или переход на новую страницу;

\n – New line или переход на новую строку;

\r– Carriage return или возврат каретки;

\t – Tab или табуляция.

Если в строку необходимо включить символ двойных кавычек, то исходную строку надо заключить в одинарные кавычки, и, наоборот, при включении внутрь строки одинарных кавычек – строка должна быть заключена в двойные кавычки.

```
alert('"Нет" любимый ответ лодыря');
alert("'Нет' любимый ответ лодыря");
```

Если есть необходимость включить в строку кавычки того же типа, в которые обрамлена строка, то в этом случае символы кавычек внутри строки должны экранироваться слэшем. Сам символ обратного слэша \ является служебным, поэтому всегда экранируется \\:

```
alert('\'Нет\' любимый ответ лодыря');
alert("\"Нет\" любимый ответ лодыря");
alert("\"Нет\"\\\"Да\" какой любимый ответ лодыря?");
```

Также в строку могут быть включены любые символы через их код:

\XXX символ заданный тремя 8-ричными цифрами XXX в диапазоне от 0 до 377;

\xXX символ заданный двумя 16-ричными цифрами XX от 00 до FF;

\uXXXX символ Unicode заданный четырьмя 16-ричными цифрами.

```
alert("\251 – знак копирайта");
alert("\xA9 – знак копирайта");
alert("\u00A9 – знак копирайта");
```

Если необходимо включить в строку вычисляемое выражение, то можно использовать обратные кавычки. Включаемые выражения должны быть обернуты в ${…}:

```
function salary(hourSalary, hour) {
  return hourSalary * hour;}
`Почасовая оплата считается как произведение стоимости часа на число
отработанных часов. Ваша сумма к выплате = ${salary(1, 2)}.` //
"Почасовая оплата считается как произведение стоимости часа на число
отработанных часов. Ваша сумма к выплате = 2."
```

Строки с обратными кавычками могут занимать более одной строки.

```
let pasport = `Стихи о советском паспорте
Я волком бы
          выгрыз
               бюрократизм.
К мандатам
          почтения нету.`;
alert(pasport);
```

Строки являются постоянными, после того, как строка создана, она не может быть изменена, но может быть создана новая строка.

Понятно почему строки в Js являются иммутабельными, поскольку в переменной содержится только ссылка на область константной памяти в которой находится само значение строки. При выполнении присваивания одной переменной другой переменной происходит копирование значения. Для строк – копируется фактически только указатель на строку в памяти, а не сама строка.

1.4.7. Логический тип (boolean)

Логические переменные могут принимать всего два значения true (истина) и false (ложь).

```
let check = true; // флажок на форме отмечен галочкой
check = false; // флажок на форме не содержит галочки
```

1.4.8. null и undefined

Нулевой тип включает только значение null. Это не отдельный тип, а специальное значение. Считается, что у значения ***null*** объектный тип, и оно говорит об отсутствии объекта.

Undefined – еще одно специальное значение, используемое в JavaScript. Оно возвращается при обращении либо к переменной, которая была объявлена, но которой никогда не присваивалось значение, либо к свойству объекта, которое не существует.

```
typeof(undefigned) // "undefined"
let x; typeof(x); // "undefined"
y=5; typeof(y.prop); // "undefined"
```

Специальное значение undefined – это не то же самое, что null. Хотя значения null и undefined не эквивалентны друг другу, оператор эквивалентности == считает их равными

```
x.prop==null // true
```

Когда требуется отличить null от undefined, используется оператор идентичности === или оператор typeof*()*.

```
x.part===null // false
```

1.4.9. Symbol

Symbol (Символ) — это примитивный тип данных, экземпляры которого уникальны и неизменяемы. В некоторых языках программирования символы также называются атомами. Функция Symbol() создаёт экземпляры типа symbol. Единственное разумное использование — сохранить символ, а затем использовать сохранённое значение для создания свойства объекта.

```
let handlerMethod  = Symbol();
this[handlerMethod] = function() {...};
```

1.4.10. Объекты

Объект в JavaScript можно понимать как коллекцию свойств, ассоциативный массив или список, состоящий из пар ключ-значение. Причем ключом может быть только строка, даже у элементов массива.

В JavaScript объекты бывают нескольких типов:

пользовательские объекты (литералы, результат работы конструктора в class)

объекты базового языка, или объекты, определяемые спецификацией ECMAScript. Например, Array, String, Date, Number, Function, Boolean, Math и т.д..

объекты, входящие в модель DOM (Document Object Model или представление документа в виде дерева тегов.), т.е. отвечающие тому, что содержится или происходит на веб-странице в окне браузера. Например, window, document, location, navigator.

Объект может быть создан с использованием литерального синтаксиса, когда в фигурных скобках через запятую перечисляются пары: имя свойства и значение. Такие объектв называются часто ассоциативными массивами. После имени свойства ставится двоеточие и затем указывается значение свойства.

```
 address = {city:"Минск", street:"проспект Жукова", house:50,
apartment:11};
```

Пустой объект может быть создан с явным и неявным использованием конструктора.

```
 ob1 = new Object();
 ob2 = {};
```

Основные операции над объектами: добавление новых свойств, изменение уже существующих свойств, удаление свойств и обращение к свойствам.

Доступ к свойствам осуществляется через точку `address.city` или с использованием квадратных скобок `address["city"]`

Например, создадим объект student для хранения информации о студенте: фамилии, имени, отчества, адреса, номера студенческого билета и среднего балла.

```
 var student = {};
 student.name='Иванов И.И.';
 student.adress='ул. Ленина, 23-7';
 student["num"]=123456;
 student['average']=7.5;
```

Квадратные скобки используются в основном, когда название свойства находится в переменной или состоит из двух слов:

```
 var average = 'ball'
 student['average']=7.5;
```

Здесь имя свойства `'ball'` является ключом в ассоциативном массиве, по которому расположено значение 7.5.

Доступ к свойству осуществляется точно так же как и при его добавлении. Через точку или квадратные скобки.

```
 alert(student['name'] + '\n' + student["address"] + '\n' +
student.num + '\n' + student.average);
```

Если у объекта нет такого свойства, то никакой ошибки при обращении по несуществующему свойству не будет, просто вернется специальное значение undefined.

```
 alert(student.fio);
```

Можно перебрать все свойства объекта. Для этого используется специальная конструкция for..in.

```
str='';
for(var key in book){
  // key – название свойства
  // object[key] – значение свойства
str=str+book[key]+'  ';
}
alert(str);
```

Можно удалить свойство объекта. Удаляет свойство оператор `delete`.

```
delete (book.pages);
str='';
for(var key in book){
  str=str+key + ': ' + book[key]+'  ';
}
alert(str);
```

У объектов JavaScript могут быть методы. Метод объекта – это функция, которая является значением свойства.

```
let book = {autor: 'Пушкин', title: 'Евгений Онегин', pages: 132,
read : function(n){alert("Прочитано "+n+" страниц.");}};
book.read(40);
```

Метод может не просто вызываться из объекта, но иметь доступ к самому объекту, может влиять на сам объект и менять находящиеся в нем данные. Для этого используется ключевое слово this.

```
let kat = { name:'Мурзик' }
let dog = { name:'Шарик' }
myPet = function() {
  alert("Мой любимый домашний питомец – "+ (this.name ? this.name :
'безымянный') )
}
// один и тот же метод в двух объектах
kat.myPet = myPet
dog.myPet =kat.myPet;
// в этом вызове – this будет kat
//Мой любимый домашний питомец – Мурзик
kat.myPet();
// в этом вызове – this будет dog
//Мой любимый домашний питомец – Шарик
dog.myPet();
// а тут – вызывается метод глобального объекта window, у которого
нет имени
myPet();
// Мой любимый домашний питомец – безымянный
```

1.4.11. Преобразования типов данных

В JavaScript значения одного типа могут быть преобразованы в значения другого типа. Это могут быть явные и неявные преобразования.

Неявное преобразование происходит когда интерпретатор автоматически выполняет преобразование типов без участия программиста. Например, если какой-нибудь оператор ожидает получить значение определённого типа, а ему

передаётся значение другого типа, то интерпретатор автоматически попытается выполнить преобразования к нужному типу:

```
33 + " коровы" // "33 коровы". Число неявно преобразуется в строку
'6' * '6';// 36. Обе строки неявно преобразуются в числа
```

Явное преобразование – это когда преобразование выполняет сам программист. Явное преобразование иначе называют приведением типов.

```
Number('2') * parseInt("12.5"));   // 25. Явное преобразование
```

JavaScript можно представлять числа в виде строк и преобразовывать строки в числа.

1.4.12. Преобразование чисел в строки

Преобразование чисел в строки производится, например, если число используется в операции конкатенации строк, оно будет преобразовано в строку:

```
x=100;
str=' руб.';
document.write('x+str='+(x+str)+'  '+typeof(x+str)+'<br>');
```

На практике, чтобы преобразовать число в строку, достаточно сложить его с пустой строкой:

```
n_as_string = n + '';
```

Функция String(number) явно преобразовывает число number в строку.

```
y_as_string =String(y);
```

Для явного преобразования числа в строку можно для объекта number вызвать метод toString().

```
z=11;
z_as_string =z.toString();
```

При этом примитивные типы чисел автоматически преобразуются в объекты типа number, благодаря чему можно воспользоваться методом toString(). Метод toString() может принимать один необязательный аргумент, который определяет основание системы счисления, в которой записано переводимое число (от 2 до 36). Если основание системы счисления не указывается, по умолчанию она предполагается равной 10.

```
z=7;
z_as_binary_string =z.toString(2);
document.write('z_as_binary_string = '+z_as_binary_string +'  ' +
typeof(z_as_binary_string) + '<br>';
```

Метод toFixed() класса number преобразует число в строку и отображает заданное число знаков после запятой.

```
var n = 1234.56789;
n.toFixed(2);
```

Метод toExponential() класса number выполняет преобразование числа в экспоненциальную форму с одним знаком перед точкой и с заданным числом десятичных знаков после точки.

```
n.toExponential(1);
```

Метод toPrecision() класса number используется для отображения определенного числа значащих разрядов. Он возвращает строку с экспоненциальным

представлением числа, если заданного количества значащих разрядов недостаточно для точного отображения целой части числа.

```
n.toPrecision(4);
```

Все три метода toFixed(), toExponential() и toPrecision() корректно выполняют округление результата

Пример преобразования числа x=100 в строку

x=100 number
str= руб. string
x+str=100 руб. string
x_as_string = 100 string
y_as_string = 25 string
z_as_string = 7 string
z_as_binary_string = 111 string
n=1234.56789 number
n.toFixed(0)=1235 string
n.toFixed(2)=1234.57 string
n.toExponential(1)=1.2e+3 string
n.toExponential(4)=1.2346e+3 string
n.toPrecision(3)=1.23e+3 string
n.toPrecision(7)=1234.568 string

1.4.13. Преобразование строк в числа

Когда строка используется как операнд в числовом выражении, она автоматически преобразуется в число. Исключение составляет сумма операндов. В этом случае плюс воспринимается как знак конкатенации строк.

```
product = "17" * "2"; //34
sum = 17 + '2'; //"172"
```

При необходимости преобразовать строку в число достаточно вычесть из строки значение 0.

```
n = '123'-0;//123
```

Для преобразования строки в число можно воспользоваться конструктором Number(str). При этом в качестве строки str может быть представление числа в строковом формате, допускается наличие ведущих и конечных символов пробела, появление других нецифровых символов не допускается.

```
n = Number('345'); //345
n = Number(' 2.34   '); //2.34
n = Number('567x'); //NaN
```

Более гибкий способ преобразования обеспечивается функциями parseInt() и parseFloat(). Эти функции преобразуют и возвращают произвольные числа, стоящие в начале строки, игнорируя любые нецифровые символы, расположенные после числа. Функция parseInt() выполняет только целочисленное преобразование. А parseFloat() может преобразовывать как целые, так и вещественные числа. Если строка начинается с символов 0x или 0X, функция parseInt() интерпретирует строку как шестнадцатеричное число. В качестве второго аргумента функция parseInt() может принимать основание системы счисления (числа в

диапазоне от 2 до 36). Если методы parseInt() и parseFloat() не могут выполнить преобразование, они возвращают значение *NaN*.

```
parseFloat("3.14 метров"); // 3.14
parseInt("3 орешка для Золушки"); // 3
parseInt("12.34"); // 12
parseInt("0xFF"); // 255
parseInt("077",8); // 63
parseInt("077",10); // 77
parseInt("two"); // NaN
parseInt("$1"); // NaN
```

1.4.14. Преобразование логических значений

Булевские или логические значения автоматически преобразуются в значения других типов. В какой тип происходит трансформация зависит от того, в каком контексте используется логическое значение. Если логическое значение используется в числовом контексте, тогда значение true преобразуется в 1, а false – в 0.

```
y=5+true; // y=6
y=1-false; // y=1
```

Если логическое значение используется в строковом контексте, тогда значение true преобразуется в строку 'true', а false – в строку 'false'.

```
'логическое значение истина ' + true ; //'логическое значение истина true'
```

1 равна true, а 0 равно false. Любое другое значение не равно, ни true, ни false.

```
1==true // true
1==false // false
0==true // false
0==false // true
17==true // false
17==false // false
NaN==true // false
NaN==false // false
```

Если в качестве логического значения используется число, оно преобразуется в значение true, если оно не равно значениям 0 или NaN, которые преобразуются в логическое значение **false**.

```
1&&true//true
17&&true//true
0&&true//0
NaN||true//true
NaN||false//false
```

Если в качестве логического значения используется не пустая строка, то она преобразуется в значение true, пустая строка преобразуется в false.

```
"abc"&&true//true
```

Специальные значения null и undefined преобразуются в false, а любые функция, объект или массив, значения которых отличны от null, преобразуются в true.

Для явного преобразования используется функция Boolean(). Результат преобразований функции Boolean():

Следующие значения в результате преобразования дают значение false: undefined, null, 0, -0, NaN, "".

Значение false возвращается без изменения.

Все остальные значения в результате преобразования дают значение true.

```
Boolean(0&&true) // false
Boolean(""&&true) // false
```

Другой способ явного преобразования заключается в использовании двойного оператора логического отрицания:

```
!!12; // true
!!0 // false
!!NaN // false
```

1.4.15. Преобразование специального значения null

Если значение null используется в логическом контексте, оно преобразуется в false, в числовом контексте оно преобразуется в значение 0, а в строковом контексте — в строку "null".

```
null&&true // null
Boolean(null&&true) // false
null||false // false
'null-12=' + (null-12) // "null-12=-12"
'специальное значение ' + null // "специальное значение null"
```

1.4.16. Преобразование специального значения undefined

Если значение undefined используется в логическом контексте, оно преобразуется в значение false. В числовом контексте – в значение NaN, а в строковом – в строку "undefined".

```
undefined&&true // undefined
Boolean(undefined && true) // false
undefined-12 // NaN
"специальное значение " + undefined // "специальное значение
undefined"
```

1.4.17. Операторы JavaScript

В таблице перечислены операторы JavaScript в порядке убывания приоритета.

приоритет оператора	Оператор	Типы операндов	Выполняемая операция
20	()	Выражение	Скобки для группировки в выражениях
19	.	Объект	Обращение к свойству
	[]	Массив, целое число	Индексация массива
	()	Функция, аргументы	Вызов функции
	new	Вызов конструктора	Создание нового объекта

приоритет оператора	Оператор	Типы операндов	Выполняемая операция
18	++	Выражение	Постфиксный инкремент
	--	Выражение	Постфиксный инкремент
17	++	Выражение	Префиксный инкремент
	--	Выражение	Префиксный инкремент
	-	Число	Унарный минус
	+	Число	Унарный плюс
	~	Целое число	Поразрядное инверсия
	!	Логическое значение	Логическое отрицание
	delete	Левостороннее значение	Аннулирование определения свойства (унарный)
	typeof	Любой	Возвращает тип данных (унарный)
	void	Любой	Возвращает неопределенное значение (унарный)
16	**	Числа	Возведение в степень
15	*, /, %	Числа	Умножение, деление, целочисленный остаток от деления
14	+, -	Числа	Сложение, вычитание
	+	Строки	Конкатенация строк
13	<<	Целые числа	Сдвиг влево
	>>	Целые числа	Сдвиг вправо с расширением знакового разряда
	>>>	Целые числа	Сдвиг вправо с дополнением нулями
12	<, <=	Числа или строки	Меньше, меньше либо равно
	>, >=	Числа или строки	Больше, больше либо равно
	instanceof	Объект, конструктор	Проверка на принадлежность к определенному типу
	in	Строка, объект	Проверка вхождения в объект
11	==	Любой	Проверка на равенство (допускается преобразование типов)
	!=	Любой	Проверка на неравенство
	===	Любой	Проверка на идентичность (не допускается преобразование типов)
	!==	Любой	Проверка на не идентичность
10	&	Целые числа	Поразрядная операция И
9	^	Целые числа	Поразрядная операция исключающего ИЛИ

приоритет оператора	Оператор	Типы операндов	Выполняемая операция
8	|	Целые числа	Поразрядная операция исключающего ИЛИ
7	&&	Логические значения	Логическое И
6	||	Логические значения	Логическое ИЛИ
5	??	Оператор нулевого слияния	логический оператор, возвращает значение правого операнда когда значение левого операнда равно null или undefined, в противном случае возвращает значение левого операнда
4	?:	Логическое значение, любое, любое	Условный тернарный оператор
3	=	Левостороннее значение, любое	Присваивание
	*=, /=, %=, +=,-=, <<=, >>=, >>>=, &=, ^=, |=	Левостороннее значение, любое	Присваивание с операцией
2	yield	любое	Функция паузы используется для остановки и возобновления функций-генераторов
1	,	Любой	Множественное вычисление выражений

В JavaScript оператор void вычисляет переданное ему выражение. При этом, независимо от того, какое именно выражение вычисляется, void всегда возвращает undefined.

```
let x = void 12; // x === undefined
```

Использование void позволяет вернуться из функции без возврата какого-либо значения (undefined), но с вызовом, например, новой функции (коллбэка):

```
// возврат чего-нибудь кроме undefined приведёт к аварийной
остановке приложения
function foo(Callback) {
  if(condition()) {return void Callback();}}
5**2 // 25
```

Оператор «запятая» предоставляет возможность последовательно вычислять несколько выражений, разделенных запятой. Каждое выражение выполняется, но возвращается результат только последнего.

```
let a = (10 + 2, 33 + 14);
alert( a ); // 47 (результат вычисления 33 + 14)
```

Такой код как в предыдущем примере не имеет смысла, но оператор «запятая» широко используется в составе более сложных конструкций, чтобы сделать код компактнее, например, в операторе for. Ниже приведен код для нахождения и вывода первых десяти чисел Фибоначчи.

```
 document.write(0 + "  " + "1" + " ");
   /*Начиная с третьего,
     находим числа Фибоначчи в цикле, */
 for(x1 = 0, x2 = 1, i = 3; i < 11 ; i++){
   x3 = x1 + x2;
   document.write(x3 + "  ");
   x1 = x2;
   x2 = x3;}
```

В первом разделе заголовка цикла записаны и выполняются последовательно три оператора. Используя оператор «запятая» этот же цикл код можно записать таким образом, чтобы тело цикла было пустым. Но можно, не значит лучше, особенно в плане легкости восприятия.

```
 for(x1 = 0, x2 = 1, i = 3; i < 11 ;x3 = x1 + x2, document.write(x3 +
"  "), x1 = x2, x2 = x3, i++){}
```

1.4.18. Сравнение

Операторы сравнения:

<, > - больше/меньше: `a > b`, `a < b`;

<=, >= - больше/меньше или равно: `a >= b`, `a <= b`;

== (двойной знак равенства) - равно: `a == b`;

!= - не равно:`a != b`.

Все операторы сравнения возвращают значение логического типа: true, либо false.

Например:

```
 alert( 17 > 10 );  // true
 alert( 10 == 17 ); // false
 alert( 10 != 17 ); // true
```

Результат сравнения можно присвоить переменной, как и любое значение:

```
 let result = 5 > 4; // результат сравнения присваивается переменной
result
 alert( result ); // true
```

Строки сравниваются в лексикографическом порядке посимвольно. Следует учитывать, что при сравнении используется кодировка Unicode, а не алфавит

При сравнении значений разных типов JavaScript приводит каждое из них к числу.

```
 '21' > 1 // true, строка '2' становится числом 2
 '010' == 10 // true, строка '01' становится числом 1
 Логическое значение true становится 1, а false – 0.
 true == 1 // true
 false == 0 // true
```

Строгое сравнение

При сравнении значений разного типа они преобразуются к числам. Это может стать проблемой. Например, при сравнении нельзя отличить число ноль от false:

```
0 == false // true
```

Аналогично, при сравнении пустой строки и false возвращается true.

```
'' == false // true
```

В этих примерах и пустая строка, и false преобразуются к нулю и только потом происходит сравнение. Чтобы исключить подобные смысловые ошибки используют оператор строгого равенства === (три равно), который проверяет на равенство без приведения типов. В этом случае если a и b имеют разные типы, то проверка a === b вернет false.

```
0 === false // false
```

Сравнение с null и undefined определены особые правила.

При строгом равенстве === эти значения различны, так как различны их типы.

```
null === undefined // false
```

При нестрогом равенстве == эти значения равны друг другу и не равны никаким другим значениям. Это специальное правило языка.

```
null == undefined // true
```

При использовании математических операторов и других операторов сравнения < > <= >= значения null/undefined преобразуются к числам: null становится 0, a undefined – NaN.

```
null > 0 // false
null == 0 // false
null >= 0 // true
```

С точки зрения математики это странно. Результат последнего сравнения говорит о том, что "null больше или равно нулю", тогда результат одного из сравнений выше должен быть true, но они оба ложны. Причина в том, что нестрогое равенство и сравнения > < >= <= работают по-разному. Сравнения преобразуют null в число, рассматривая его как 0. Поэтому выражение null >= 0 истинно, a null > 0 ложно. С другой стороны, для нестрогого равенства == значений undefined и null действует особое правило: эти значения ни к чему не приводятся, они равны друг другу и не равны ничему другому. Поэтому null == 0 ложно.

Значение undefined несравнимо с другими значениями:

```
undefined > 0 // false
undefined < 0 // false
undefined == 0 // false
```

Первые два сравнения возвращают false, потому что undefined преобразуется в NaN, a NaN – это специальное числовое значение, которое возвращает false при любых сравнениях. Третье сравнение возвращает false, потому что undefined равно только null, undefined и ничему больше.

Нужно быть осторожным при использовании операторов сравнений вроде > и < с переменными, которые могут принимать значения null/undefined. Лучше использовать отдельную проверку на null/undefined.

1.4.19. if … else, else if

Условный оператор if … else имеет вид такой же, как и в java и C++.

```
if (условие)
оператор 1;
[else оператор 2;]
if (условие){
  блок кода 1
}
[else {
   блок кода 2
 }]
```

Если в случае выполнения (или не выполнения) условия необходимо выполнить группу операторов, то они берутся в фигурные скобки. else может отсутствовать.

Если необходимо проверить несколько уточняющих условий с каждым из которых связан блок кода можно воспользоваться лесенкой else if. Формально это не отдельный оператор JavaScript, а распространенный стиль программирования, состоящий в применении повторяющихся инструкций if … else.

```
if (n == 1) {
// блок кода 1
}
else if (n == 2) {
// блок кода 2
}
else if (n == 3) {
// блок кода 3
}
else {
// Если все остальные условия else не выполняются, исполняем блок
кода 4
}
```

Лесенка else if – это просто последовательность инструкций if, где каждая инструкция if является частью конструкции else предыдущей инструкции. Стиль else if предпочтительнее и понятнее записи в синтаксически эквивалентной форме.

```
if (n == 1) {
// блок кода 1
}
else {
if (n == 2) {
// блок кода 2
}
else {
if (n == 3) {
// блок кода 3
```

```
 }
 else {
 // Если все остальные условия else не выполняются, исполняем блок
кода 4
 }
 }
 }
```

Условный оператор `if (…)` вычисляет выражение в скобках и преобразует результат к логическому типу.

Напомним, что согласно правилу преобразования типов:

число 0, пустая строка "", null, undefined и NaN преобразуются к false.

остальные значения преобразуются к true.

Например, кусок кода в фигурных скобках никогда не выполнится `if (0) { … }`, а при таком условии `if (1) { … }` – будет выполняться всегда.

1.4.20. Тернарный оператор

Тернарный оператор условия

переменная = (условие) ? значение1 ***:*** значение2;

Если выполняется условие, то а переменную заносится значение1, иначе – значение2. Круглые скобки не обязательны, но делают код читабельнее.

1.4.21. switch

Оператор выбора switch.

```
 switch (переменная или выражение) {
 case значение1: // Выполняется, если переменная или выражение
совпадает со значением1
 операторы блока кода 1
 break; // остановка и выход за пределы switch
 case значение2: // Выполняется, если переменная или выражение
совпадает со значением2
 операторы блока кода 2
 break; // остановка и выход за пределы switch
 …
 case значениеN: // Выполняется, если переменная или выражение
совпадает со значениемN
 операторы блока кода N
 break; // остановка и выход за пределы switch
 [default: // Выполняется, если переменная или выражение не совпадает
ни с одним из перечисленных в case значением
 операторы блока кода N+1
 break;] // остановка и выход за пределы switch
 }
```

Оператор switch сначала вычисляет значение выражения, а затем переходит на case, соответствующий этому значению. Если case найден, исполняется блок кода, начиная с первого оператора, следующего за case. Если соответствующий case не найден, то выполняется блок кода, расположенный за default. Если метки default нет, то блок кода пропускается целиком. Ключевое слово break в конце каждого блока case приводит к передаче управления в конец ин-

струкции switch. Конструкции case в switch задаёт только начальную точку исполняемого кода, но не задают конечной точки. В случае отсутствия break оператор switch начинает исполнение блока кода с case, соответствующего значению выражения, и продолжает исполнение до тех пор, пока не дойдет до конца блока (не встретит break или не выполнит весь код в switch. При использовании switch внутри функции можно вместо break использовать return.

Приведем пример использования оператора switch. Функция convert преобразует значение в строку способом, зависящим от типа значения: число преобразуется в строку содержащую шестнадцатеричную запись числа; к строке дописываются кавычки, а логическое значение записывается в верхнем регистре.

```
<script>
function convert(x) {
switch(typeof(x)) {
case 'number': // Преобразуем число в шестнадцатеричное целое
return x.toString(16);
case 'string': // Возвращаем строку, заключенную в кавычки
return '"' + x + '"';
case 'boolean': return x.toString().toUpperCase();// Преобразуем в
TRUE или FALSE, в верхнем регистре
default: // Любой другой тип преобразуем обычным способом
return x.toString()
}
}
document.write(convert(200)+'<br>');
document.write(convert('qwerty')+'<br>');
document.write(convert(true));
</script>
```

В качестве результата получим:

```
c8
"qwerty"
TRUE
```

1.4.22. цикл while

Цикл с предусловием с неизвестным количеством повторений представлен оператором while. Синтаксис оператора цикла с предусловием while совпадает с синтаксисом C++ и Java.

```
while (условие){
```

оператор;}

Чтобы цикл не продолжался вечно, условие должно изменяться, например, в теле цикла.

```
let i = 3;
while (i) { // когда i будет равно 0, условие станет ложным, и цикл
остановится
   let years=prompt("Вам даётся три попытки ввода ответа. Попытка " +
(3-i+1) + ":\nСколько дней в високосном году?",'360');
   if(years==366){alert("Правильно! Угадали с " + (3-i+1) + "
попытки."); break;}
   i--;
```

```
}
if(i==0)alert("Стыдно не знать таких вещей!");
```

1.4.23. цикл do while

Для организации цикла с предусловием существует оператор

```
do
  оператор;
while (условие);
```

Этот цикл проверяет условие продолжения после выполнения цикла, поэтому тело цикла выполняется как минимум один раз, независимо от того, изначально условие выполняется или нет. Ниже приведен код примера игры, где пользователь может проверить свою удачу. Проверка удачливости в любом случае проверяется по крайней мере один раз.

```
alert(" Проверим кто удачливее! Кто задумает большее число от 0 до
1, тот и выиграл.");
let myWin=0, yourWin=0, str, yourAnswer;
do {
  let myNumber = Math.random()*10;
  let yourNumber=prompt("Какое число задумал ты?");
  if(yourNumber>myNumber)
    {str = "Ты победил!";yourWin++;}
  else if (yourNumber===myNumber)
    {str = "Ничья!";yourWin++;myWin++;}
  else {str = "Я выиграл!";myWin++;}
  yourAnswer=prompt(str + "Продолжим (да/нет)?");
} while(yourAnswer == "да");
let win=yourWin-myWin;
if(win) {str = "Тебе везет чаще!"}
else if (yourNumber===myNumber)
  {str = "Удача любит нас одинаково!";}
else {str = "Мне везет чаще!";}
alert(str);
```

1.4.24. цикл for

Цикл с известным количеством повторений for.

```
for (блок инициализации; проверка условия; изменение счетчика цикла)
оператор;
```

Сначала один раз выполняется блок инициализации, затем проверяется условие, выполняется оператор тела цикла, изменяется счетчик и снова проверяется условие, выполняется оператор тела цикла, изменяется счетчик и т д.

```
let pet = ["кот", "собака", "попугай", "хомяк"];
for (var i = 0; i < pet.length; i++){
  alert(pet[i] + " – это домашнее животное.");}
```

1.4.25. цикл for in

Оператор цикла for in используется для перебора в случайном порядке перечисляемых свойств объекта

```
for (переменная in объект)
```

```
оператор;
```

Например, просмотреть все свойства объекта document можно следующим образом:

```
for(prop in document)
document.write("имя: " + prop + "; значение: " +document[prop] +
"<br>");
```

При помощи ***for in*** можно легко скопировать имена всех свойств объекта в массив.

```
let ob = {x:1, y:2, z:3};
let a = new Array();
let i = 0;
for(a[i++] in ob); // тело цикла  пусто
for(prop in a)
document.write("a[" + prop + "] = " + a[prop] + "<br>");
```

1.4.26. continue

Позволяет перейти к следующей итерации цикла, еще до завершения выполнения всех операций предыдущей итерации. Может использоваться в любом цикле.

Например, приведенный ниже код использует continue в цикле for, чтобы выводить только нечётные значения.

```
for (let i = 0; i < 10; i++) {
  // если true, пропустить оставшуюся часть тела цикла
  if (i % 2 == 0) continue;
  alert(i);}
```

Для чётных значений i, директива continue прекращает выполнение текущей итерации и управление передаётся проверке условия цикла. Если условие цикла true, то выполняется следующая итерация и т.д. Таким образом alert вызывается только для нечётных значений.

1.4.27. break

Специальная директива break приводит к выходу из цикла или из операторов switch. JavaScript допускает указание имени метки за ключевым словом ***break***.

```
break: имя_метки;
```

Когда break используется с меткой, происходит переход на именованный оператор, внешний по отношению к break. Между ключевым словом break и именем метки перевод строки не допускается. Если разбить строку кода между ключевым словом break и следующей за ним меткой, интерпретатор JavaScript предположит, что имелась в виду простая форма break без метки, и добавит точку с запятой.

Директива break часто используется в бесконечных циклах, в которых условие, по которому нужно прерваться, находится в теле цикла. В следующем примере ведется подсчет суммы чисел, которые вводит пользователь, Числа суммируются до тех пор, пока посетитель их вводит, а затем выводится результат.

```
let sum = 0;
while (true) {
  let value = +prompt("Введите число", '');
  if (!value) break; // (*)
  sum += value;
}
alert( 'Сумма: ' + sum );
```

1.4.28. Деструктуризация объектов/массивов

Деструктуризация — это синтаксис, который позволяет распаковывать значения из массивов или свойства из объектов в переменные.

Что нам это дает? Возможность писать более чистый и понятный код, экономя время и количество строк.

(https://habr.com/ru/company/skillbox/blog/658007/)

```
const customer = {
name: 'Sherlock',
email: 's.h.@gmail.com',
age: 34,
address: {
  streetNo: '221b Baker Street',
  city: 'London',
  country: 'UK'
 }
}
```

Было так

```
const name = customer.name;
const email = customer['email']; // или так
const streetNo = customer.address.streetNo;
стало возможно писать короче - так:
const { name, email } = customer;
const { name: customerName, email } = customer;//присваивание в переменную с другим именем
const { name, address: { streetNo, city } } = customer;
```

Присвоение значений уже объявленным переменным:

```
let name, email; ({ name, email } = customer);//
```

1.4.29. Логические операторы

Логические И `&&`, ИЛИ `||` используют так называемое «ленивое вычисление» и возвращают значение, на котором оно остановилось (не обязательно `true` или `false`). Логическое НЕ `!` конвертирует операнд в логический тип и возвращает инвертированное значение.

Очень часть можно встретить такие выражения: `value || 'default value'` - Указание значения, используемого по умолчанию. На практике часто требуется использовать значение по умолчанию только тогда, когда переменная является `null/undefined`. Ведь именно тогда значение действительно неизвестно/не определено.

Проблема этого кода заключается в том, что появление тут числа `0`, значения `false` или пустой строки будет распознано как значение `false`, а это — не

то, что нам нужно. Именно поэтому проверку на `null` и `undefined` нужно проводить в явном виде:

```
value !== null && value !== undefined
```

В ES6 есть оператор, называемый «объединение со значением `null`» (nullish coalescence), который выглядит как два знака вопроса (`??`). В подобной ситуации теперь можно будет воспользоваться следующей конструкцией: `value ?? 'default value';` Это защищает нас от случайного использования значения по умолчанию при появлении в выражениях значений, распознаваемых как `false`, но при этом позволяет выявлять значения `null` и `undefined`, не прибегая к тернарному оператору и к проверке вида `!= null`.

Раньше многие были недовольны оператором ||. Важное различие между || и ?? заключается в том, что: || возвращает первое истинное значение. ?? возвращает первое определённое значение. Довольно часто в реальных проектах нам приходится видеть многочисленные проверки на корректность данных - строки примерно такого вида: перед обращением к полю `data.user.address.street` приходится проверять, что объект имеет структуру, которую мы от ожидаем - `const street = data && data.user && data.user.address && data.user.address.street;` иначе будет runtimeError Uncaught TypeError: Cannot read property 'street' of undefined
В ES6 можно использовать optional chaining `data.user?.address?.street`

1.4.30. Операторы: запятая, три точки (spread, rest)

а) оператор `...` *(rest)*. **http://code.mu/ru/javascript/book/prime/ellipsis/rest/.**

Если `...` *rest* стоит в конце списка параметров функции, то он **создаёт массив** из списка параметров

```
function sumAll(...args) { // args — имя массива
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;
}
[a,..b]=[1,2,3,4];//a=1, b=[2,3,4];
```

б) оператор `...` *(spread)*. **http://code.mu/ru/javascript/book/prime/ellipsis/spread/**

Если `...` spread стоит **перед** массивом, он **разделяет этот массив** на отдельные значения, превращая массив в набор параметров!

Если некая функция ожидает получить список чисел, а не один массив, то `...` извлекает элементы из массива.

spread работает с любым перебираемым объектом, а не только с массивом!

Под капотом оператор расширения использует итераторы, чтобы перебирать элементы. Так же, как это делает `for..of`. Поэтому, например, для строк тоже работает

```
let arr1 = [1, 2, 3, 4, 5], arr2=[22,2];
let max = Math.max(...arr1,13,4,5,...arr2);
let rezult=[...arr1,..arr2];//сливаем массивы!!!
```

```
let str="Hello!";
arr2=[...str];//символы из строки собираются в массив
/* АНТИПАТТЕРН!!! [1,2,3,4,5].reduce((acc, value, index) => ({...acc,
[value]: index}), {}); */
```

1.4.31. Функции

Три способа создания **функции** в JavaScript:

1. **Function Declaration:** функция в основном потоке кода

```
 function sum(a, b) {
let result = a + b;
return result; }
```

2. **Function Expression**: функция как часть любого выражения
   let sum = function(a, b) { let result = a + b; return result; };
3. Стрелочные функции:

```
 // некий код в правой части выражения
let sum = (a, b) => a + b;
// многострочный код в фигурных скобках { ... }, здесь нужен return:
let sum = (a, b) => { // ... return a + b; }
// без аргументов
let sayHi = () => alert("Привет");
// с одним аргументом
let double = n => n * 2;
```

- У функций могут быть локальные переменные: т.е. объявленные в теле функции. Такие переменные видимы только внутри функции.
- У параметров могут быть значения по умолчанию: `function sum(a = 1, b = 2) {...}`.
- Функции всегда что-нибудь возвращают. Если нет оператора `return`, результатом будет `undefined`.

В программировании есть правило: «Программный код, который используется два и более раз должен быть оформлен в функцию».

Функции в JavaScript можно рассматривать как специальный тип объекта, т. е. функция – это объект, с которым связан исполняемый код, который определен в JavaScript программе или в интерпретаторе JavaScript. Функции могут быть встроенными, например, в коде ниже вызывается функция sin, стандартного встроенного объекта Math.

```
 document.write('sin(0)=' + Math.sin(0)); // sin(0)=0
```

Есть пользовательские функции. Например, для определения максимального из двух чисел можно создать следующую функцию:

```
 function max(x,y){
   return x>y ? x : y;
 }
 document.write('max(8,-3)=' + max(8,-3) + '<br>'); // max(8,-3)=8
```

В JavaScript функции представляют собой тип данных, их можно определять и вызывать. Поскольку функции – это тип данных, то они представляют собой значения и могут храниться в переменных, массивах и объектах, а также передаваться в качестве аргументов другим функциям.

Чтобы создать функцию ее надо объявить. Объявить функцию можно несколькими способами. Первый способ еще называют классическим. Вначале записывают ключевое слово function, после него имя функции, затем в круглых скобках через запятую перечисляют список параметров, после списка параметров в фигурных скобках помещают тело функции. Возвращаемое значение помещают в оператор return.

```
function имя(параметры) {
тело
return результат}
```

Если функция ничего не возвращает, то return можно опускать. Если использовать return без значения, то это приведёт к немедленному выходу из функции. Отсюда следует, что нельзя размещать return и возвращаемое значение на разных строках.

```
function Hello(person) {
  alert("Hello, "+ person);
}
```

Программный код, расположенный в теле функции, выполняется не в момент объявления функции, а в момент её вызова. При вызове указывается имя функции и параметры.

```
Hello("Маша")
```

Функции, объявленные первым, классическим способом, создаются интерпретатором до начала выполнения кода, поэтому их можно вызывать ещё до объявления, но только в области их видимости.

```
activWork();// Ошибка!
Hello("Маша"); // Ошибки не будет
function Hello(person) {
  alert("Hello, "+ person);
    activWork();
  function activWork() {
    alert("Работай активнее!");  }
}
```

Так, например, вызов функции activWork() первый раз приведет к ошибке, поскольку вызов производится вне области видимости функции. А вот вызов функции Hello() до объявления происходит в области ее видимости, поэтому ошибки не вызовет. Аналогично, вызов activWork до объявления внутри функции Hello не вызовет ошибки по той же причине.

Второй способ объявления функций – это когда объявление функции является частью какого-либо выражения, например присваивания.

```
Hello = function (person) {alert("Hello, "+ person);}
Hello("Маша");
```

В этом случае функции создаются в момент выполнения кода, поэтому их нельзя вызывать до их объявления.

Переменные, объявленные внутри функции с помощью ключевого слова let, являются локальными, и доступны только внутри этой функции.

Внутри функций есть доступ к внешним переменным и можно изменить значение внешней переменной прямо внутри функции:

```
let person = "Маша";
function Hello() {
```

```
   alert("Hello, "+ person);// Маша
   person = "Катя"; // изменяем значение внешней переменной
   let message = 'Привет, ' + person;
   alert(message);
 }
 alert(person); // person – Маша перед вызовом функции

 Hello();
 alert(person); // person – Катя, значение внешней переменной было
изменено функцией
```

Параметры передаются в функцию копией, поэтому изменение значений параметров внутри функции не приведет к их изменению вне функции.

```
 function Hello(person) {
   person = "Катя"; // изменяем значение параметра
   alert("Hello, "+ person);// Hello, Катя
 }
 let person = "Маша";
 alert(person); // Маша перед вызовом функции
 Hello(person);
 alert(person); // Маша, значение внешней переменной осталось прежним
```

Если параметр не указан, то его значением становится undefined.

Если функция вызывается как метод объекта, то параметр this – это ссылка на объект, который производит вызов функции. Если функция, не является методом объекта, тогда this равняется undefined.

Напишем функцию, которая будет находить корни квадратного уравнения и выводить результат в окно с сообщением и кнопкой Ok. Обратите внимание, что рассмотрены все случаи: квадратное уравнение, линейное уравнение, и вырожденные случаи.

```
 function solveQuadratic(a, b, c) {
   // Проверка, что все три коэффициента не равны нулю
     if (a !== 0){ // Если уравнение квадратное
       // Вычисление дискриминанта
       let d = b ** 2 - 4 * a * c;
       // Проверка, что дискриминант не отрицательный
       if (d < 0) {
       alert('Уравнение квадратное. Действительных корней нет.');
       return;
       }//конец d < 0
       // Вычисление корней
       let x1 = (-b + Math.sqrt(d)) / 2 / a;
       let x2 = (-b - Math.sqrt(d)) / 2 / a;
       alert('Уравнение квадратное. Действительные корни x1 = '
+`${x1}` + ', x2 = ' + `${x2}` + '.' );
       return;
   }//конец a !== 0
   else{ //a=0, уравнение не квадратное, а линейное
       if(b!==0){//a=0, b!=0, коэффициент при x не нулевой, уравнение
линейное
         let x = -c / b;
         alert('Уравнение линейное. Имеет один корень x = ' + `${x}`
+ '.');
```

```
          return;
        }//конец a=0, b!=0
        else {
            if( c === 0 ){//a=0, b=0, c=0 все коэффициенты нулевые,
решение - вся числовая прямая
                alert('Все коэффициенты нулевые, решение - вся
числовая прямая.');
                return;
            }//конец //a=0, b=0, c=0
            else{//a=0, b=0, c!=0 коэффициенты при степенях переменной
нулевые, а свободный член не нулевой, решений нет
              alert('Все коэффициенты при степенях переменной нулевые,
а свободный член не нулевой, решений нет.');
              return;
            }//a=0, b=0, c!=0
        }//конец a=0, b=0
    }//конец a=0

 }// конец function Quadratic
  solveQuadratic(100, -2, 1)
```

Решение находится, но не записывается в переменную для дальнейшего использования. Будем возвращать из функции объект. Сразу при входе в функцию создадим пустой объект без свойств

```
let res = {};
```

Затем во всех местах, где методом alert выводилось окно с сообщением добавим в объект rez свойство info и заполним его.

```
function solveQuadratic(a, b, c) {
     let res = {};// создаем пустой объект без свойств
   // Проверка, что все три коэффициента не равны нулю
     if (a !== 0){ // Если уравнение квадратное
       // Вычисление дискриминанта
       let d = b ** 2 - 4 * a * c;
       // Проверка, что дискриминант не отрицательный
       if (d < 0) {
           res.info = 'Уравнение квадратное. Действительных корней
нет.';
          // alert('Уравнение квадратное. Действительных корней
нет.');
           return res;
       }//конец d < 0
       // Вычисление корней
       let x1 = (-b + Math.sqrt(d)) / 2 / a;
       let x2 = (-b - Math.sqrt(d)) / 2 / a;
       res.info = 'Уравнение квадратное. Действительные корни x1 = '
+`${x1}` + ', x2 = ' + `${x2}` + '.'
       //alert('Уравнение квадратное. Действительные корни x1 = '
+`${x1}` + ', x2 = ' + `${x2}` + '.' );
       return res;
   }//конец a !== 0
   else{ //a=0, уравнение не квадратное, а линейное
       if(b!==0){//a=0, b!=0, коэффициент при x не нулевой, уравнение
линейное
```

```
        let x = -c / b;
        res.info = 'Уравнение линейное. Имеет один корень x = ' +
`${x}` + '.';
        //alert('Уравнение линейное. Имеет один корень x = ' +
`${x}` + '.');
        return res;
      }//конец a=0, b!=0
      else {
          if( c === 0 ){//a=0, b=0, c=0 все коэффициенты нулевые,
решение - вся числовая прямая
              res.info = 'Все коэффициенты нулевые, решение - вся
числовая прямая.';
              //alert('Все коэффициенты нулевые, решение - вся
числовая прямая.');
              return res;
          }//конец //a=0, b=0, c=0
          else{//a=0, b=0, c!=0 коэффициенты при степенях переменной
нулевые, а свободный член не нулевой, решений нет
            res.info = 'Все коэффициенты при степенях переменной
нулевые, а свободный член не нулевой, решений нет.'
            //alert('Все коэффициенты при степенях переменной
нулевые, а свободный член не нулевой, решений нет.');
            return res;
          }//a=0, b=0, c!=0
      }//конец a=0, b=0
  }//конец a=0

}// конец function solveQuadratic
solveQuadratic(1,1,1)
```

В результате запуска `solveQuadratic(100, -4, 1)` получим:

```
Object { info: "Уравнение квадратное. Действительных корней нет." }
    info: "Уравнение квадратное. Действительных корней нет."
```

Добавим к объекту свойство amount, которое будет содержать количество корней, если они есть, Infinity – если корнями является вся числовая прямая и NaN – если корней нет.

Если уравнение имеет корни, то добавим к объекту свойство root, которое будет содержать корни. Если корней нет, то и свойство добавлять не будем. Если уравнение имеет бесконечное число корней, то также свойство добавлять не будем.

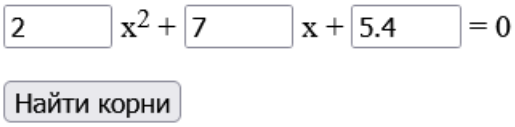
```
function solveQuadratic(a, b, c) {
    let res = {};// создаем пустой объект без свойств
  // Проверка, что все три коэффициента не равны нулю
    if (a !== 0){ // Если уравнение квадратное
      // Вычисление дискриминанта
      let d = b ** 2 - 4 * a * c;
      // Проверка, что дискриминант не отрицательный
      if (d < 0) {
          res.info = 'Уравнение квадратное. Действительных корней
нет.';
          res.amount = 0;
        // alert('Уравнение квадратное. Действительных корней
нет.');
```

```
            return res;
        }//конец d < 0
        // Вычисление корней
        let x1 = (-b + Math.sqrt(d)) / 2 / a;
        let x2 = (-b - Math.sqrt(d)) / 2 / a;
        res.info = 'Уравнение квадратное. Действительные корни x1 = '
+`${x1}` + ', x2 = ' + `${x2}` + '.'
        res.amount = 2;
        // Возвращение массива с корнями в нужном порядке с помощью
тернарного оператора
        res.root = x1 < x2 ? [x1, x2] : [x2, x1];
        //alert('Уравнение квадратное. Действительные корни x1 = '
+`${x1}` + ', x2 = ' + `${x2}` + '.' );
        return res;
    }//конец a !== 0
    else{ //a=0, уравнение не квадратное, а линейное
        if(b!==0){//a=0, b!=0, коэффициент при x не нулевой, уравнение
линейное
          let x = -c / b;
          res.info = 'Уравнение линейное. Имеет один корень x = ' +
`${x}` + '.';
          res.amount = 1;
          res.root = [x];
          //alert('Уравнение линейное. Имеет один корень x = ' +
`${x}` + '.');
          return res;
        }//конец a=0, b!=0
        else {
            if( c === 0 ){//a=0, b=0, c=0 все коэффициенты нулевые,
решение - вся числовая прямая
                res.info = 'Все коэффициенты нулевые, решение - вся
числовая прямая.';
                res.amount = Infinity;
                //alert('Все коэффициенты нулевые, решение - вся
числовая прямая.');
                return res;
            }//конец //a=0, b=0, c=0
            else{//a=0, b=0, c!=0 коэффициенты при степенях переменной
нулевые, а свободный член не нулевой, решений нет
              res.info = 'Все коэффициенты при степенях переменной
нулевые, а свободный член не нулевой, решений нет.';
              res.amount = 0;
              //alert('Все коэффициенты при степенях переменной
нулевые, а свободный член не нулевой, решений нет.');
              return res;
            }//a=0, b=0, c!=0
        }//конец a=0, b=0
    }//конец a=0

 }// конец function solveQuadratic
```

Теперь разработаем страницу, на которой выведем общий вид квадратного уравнения с полями для ввода коэффициентов и разместим кнопку, при нажа-

тии на котрую должно выводиться само уравнение, текст о наличии решения и корнях уравнения и произведение корней уравнения на их количество.




```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>Решение квадратного уравнения</title>
<script src="SquareRoot2.js"></script>
</head>

<body>

   <div>
    <input id="a" name="a" value="0" size="5"> x<sup>2</sup> +
    <input id="b" name="b" value="0" size="5"> x +
    <input id="c" name="c" value="0" size="5"> = 0

    <p><button onClick="solve();">Найти корни</button>
    </p>

   </div>
   <p id="out"></p>
</body>
</html>
Файл SquareRoot2.js со скриптами.
//Функция для вывода на экран квадратного уравнения a x^2 + b x + c
= 0
function equationToStr(a, b, c){
    let str = '';
    str= str +a + 'x<sup>2</sup> ';
    if(b>=0){
        str = str + ' +';
    }
    else {
        str = str + ' ';
    }
    str = str + b + 'x ';
    if(c>=0){
        str = str + ' +';
    }
    else str = str + ' ';
    str = str + c ;
    str= str + ' = 0';
    return str;
}

//Функция для нахождения решения квадратного уравнения
function solveQuadratic(a, b, c) {
    let res = {};// создаем пустой объект без свойств
```

```
   // Проверка, что все три коэффициента не равны нулю
     if (a !== 0){ // Если уравнение квадратное
       // Вычисление дискриминанта
       let d = b ** 2 - 4 * a * c;
       // Проверка, что дискриминант не отрицательный
       if (d < 0) {
           res.info = 'Уравнение квадратное. Действительных корней
нет.';
           res.amount = 0;
           return res;
       }//конец d < 0
       // Вычисление корней
       let x1 = (-b + Math.sqrt(d)) / 2 / a;
       let x2 = (-b - Math.sqrt(d)) / 2 / a;
       res.info = 'Уравнение квадратное. Действительные корни x1 = '
+`${x1}` + ', x2 = ' + `${x2}` + '.'
       res.amount = 2;
       // Возвращение массива с корнями в нужном порядке с помощью
тернарного оператора
       res.root = x1 < x2 ? [x1, x2] : [x2, x1];
       return res;
   }//конец a !== 0
   else{ //a=0, уравнение не квадратное, а линейное
       if(b!==0){//a=0, b!=0, коэффициент при x не нулевой, уравнение
линейное
         let x = -c / b;
         res.info = 'Уравнение линейное. Имеет один корень x = ' +
`${x}` + '.';
         res.amount = 1;
         res.root = [x];
         return res;
       }//конец a=0, b!=0
       else {
           if( c === 0 ){//a=0, b=0, c=0 все коэффициенты нулевые,
решение - вся числовая прямая
               res.info = 'Все коэффициенты нулевые, решение - вся
числовая прямая.';
               res.amount = Infinity;
               return res;
           }//конец //a=0, b=0, c=0
           else{//a=0, b=0, c!=0 коэффициенты при степенях переменной
нулевые, а свободный член не нулевой, решений нет
             res.info = 'Все коэффициенты при степенях переменной
нулевые, а свободный член не нулевой, решений нет.';
             res.amount = 0;
             return res;
           }//a=0, b=0, c!=0
       }//конец a=0, b=0
   }//конец a=0

 }// конец function solveQuadratic
```

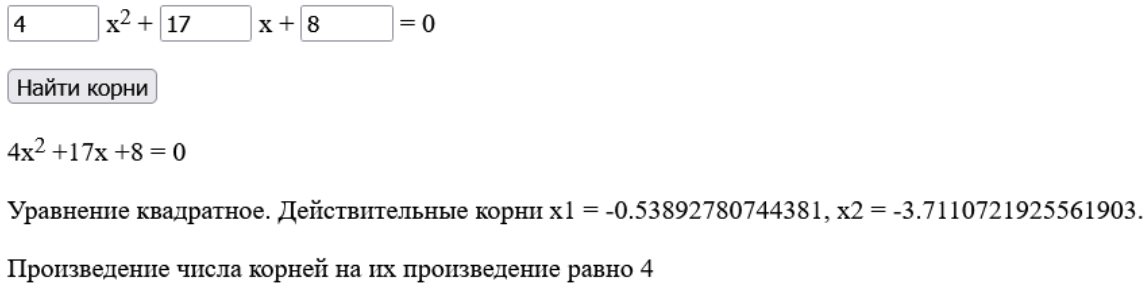
```
 //Функция получает из полей ввода коэффициенты квадратного
уравнения, решает его, и выводит требуемую информацию на страницу
 function solve(){
     let a = document.getElementById('a').value-0;//Первый вариант
преобразования в число
     let b = Number(document.getElementById('b').value);//Второй
вариант преобразования в число
     let c = parseFloat(document.getElementById('c').value);//Третий
вариант преобразования в число
     let str = equationToStr(a, b, c);
     r=solveQuadratic(a, b, c);
     let outElement = document.getElementById('out');
     //удаляем все содержимое узла
     outElement.textContent='';
     //создаем параграф для записи уравнения
     let elemEq = document.createElement('p');
     //записываем в него текст уравнения
     elemEq.innerHTML = str;
     // добавляем элемент в блок out
     outElement.appendChild(elemEq);

     let elemInfo = document.createElement('p');
     elemInfo.innerHTML = r.info;
     outElement.appendChild(elemInfo);

     //создаем параграф для записи произведения числа корней на их
количество
     let elemProd = document.createElement('p');
     if(r.amount === 2){
         str = 'Произведение числа корней на их произведение равно '
+ String(r.root["0"]*r.root["1"]*r.amount);
     }
     if(r.amount === 1){
         str = 'Произведение числа корней на их произведение равно '
+ String(r.root);
     }
      if(isFinite(r.amount) === false){
        str = 'Произведение числа корней на их произведение равно ' +
String(Infinity);
     }
     if(r.amount === 0){
        str = 'Корней нет.';
     }
     //записываем в него текст
     elemProd.innerHTML = str;
     // добавляем элемент в блок out
     outElement.appendChild(elemProd);
 }
```

Обратите внимание, что при получении данных из поля input тип данных всегда строка, независимо от того, хранятся там числа или нет. Поэтому при получении коэффициентов уравнения строки необходимо преобразовать в чис-

ла. Коэффициентов три, и каждый из них в демонстрационных целях преобразуется к числу уникальным способом. Результат работы программы:



1.4.32. Массивы в JavaScript

Массивы предоставляют множество готовых методов "из коробки" поэтому не надо реализовывать свои "велосипеды" (https://learn.javascript.ru/array-methods)

```
 let arr = new Array(); //Вызов new Array(number) создаёт массив с заданной длиной, но без элементов!!!.
let arr = [];//для массива заранее выделяется много памяти (для оптимизации)
let arr=Array.from({ length: N })//создаёт массив с длиной N и с элементами=undefined
let arr=Array.from({ length: N },()=>2)//создаёт массив с длиной N и с элементами=2
let arr=Array.from({ length: N }, () => Array(N).fill())//создаёт ДВУМЕРНЫЙ массив N*N с элементами=undefined
```

Элементы массива нумеруются, начиная с нуля. *Висячая запятая* игнорируется (`arr = [1,2,3,]`)! В массиве могут храниться элементы любого типа. Но **не стоит никогда** так делать (`arr = [1,"2",3.5,]`).

Свойство `length` отражает длину массива или, если точнее, его последний цифровой индекс плюс один. Длина корректируется автоматически методами массива.

- Если мы уменьшаем `length` вручную, массив укорачивается. **Не надо увеличивать** свойство `length` вручную! Это всегда приводит к резкому падению производительности т.к. появляются "дырки" в массиве.

Движок JavaScript старается хранить элементы массива (если они однотипные) в непрерывной области памяти, один за другим. Поэтому методы `push/pop` выполняются быстро, а методы `shift/unshift` – медленно.

Массив является объектом и, следовательно, ведёт себя как объект. Например, **копируется по ссылке!**

Копирование массивов/объектов

поверхностное `dest=[...src]; dest=src.slice(0); dest={...src}`. Работает в частности для многомерных числовых массивов. Клонирование вложенных свойств по ссылке пропускается, нужно быть осторожным в использовании.

полуглубокое - `dest=JSON.parse(JSON.stringify(src));` вариант для JSON-safe объектов в частности для многомерных числовых массивов. Опасен для рекурсивных объектов или когда имеется конструктор с параметрами

глубокое - написать собственную рекурсию с проверками всевозможными или использовать `deserialize(require("v8").serialize(obj));`.

Перебор элементов:

```
for (let i = 0; i < arr.length; i++) alert( arr[i] );
for (let fruit of fruits) alert( fruit );
```

Цикл `for..of` не предоставляет доступа к номеру текущего элемента, только к его значению, но в большинстве случаев этого достаточно и эта запись короче

- `forEach(func)` – вызывает `func(elem,i,arr)` для каждого элемента. Ничего не возвращает. Нельзя остановить выполнение этого цикла, например, если нашли какое-то условие.

Для поиска среди элементов:

- `indexOf/lastIndexOf(item, pos)` – ищет `item`, начиная с позиции `pos`, и возвращает его индекс или `-1`, если ничего не найдено. **Работает медленно! Если надо искать постоянно, то используйте Map в котором вызывайте has**
- `includes(value)` – возвращает `true`, если в массиве имеется элемент `value`, в противном случае `false`. **Работает медленно! Если надо искать постоянно, то используйте Map в котором вызывайте has**
- `find/filter(func)` – фильтрует элементы через функцию и отдаёт первое/все значения, при прохождении которых через функцию возвращается `true`.
- `findIndex` аналог `find`, но возвращает индекс вместо значения.

Для преобразования массива:

- `map(func)` – создаёт новый массив из результатов вызова `func` для каждого элемента.
- `sort(func)` – сортирует массив «на месте», а потом возвращает его.
- `reverse()` – «на месте» меняет порядок следования элементов на противоположный и возвращает изменённый массив.
- `split/join` – преобразует строку в массив и обратно. Работает для одномерных массивов.
- `JSON.parse(src)/JSON.stringify(src)` – преобразует строку в массив и обратно. Для многомерных массивов тоже работает.
- `reduce(func, initial)` – вычисляет одно значение на основе всего массива, вызывая `func` для каждого элемента и передавая промежуточный результат между вызовами.
- Дополнительно:
- `Array.isArray(arr)` проверяет, является ли `arr` массивом.

Обратите внимание, что методы `sort`, `reverse` и `splice` изменяют исходный массив.

1.4.33. Map и Set

Map – это коллекция ключ/значение, как и `Object`. Но основное отличие в том, что `Map` позволяет использовать ключи любого типа.

Методы и свойства:

- `new Map()` – создаёт коллекцию.
- `map.set(key, value)` – записывает по ключу `key` значение `value`.
- `map.get(key)` – возвращает значение по ключу или `undefined`, если ключ `key` отсутствует.
- `map.has(key)` – возвращает `true`, если ключ `key` присутствует в коллекции, иначе `false`.
- `map.delete(key)` – удаляет элемент по ключу `key`.
- `map.clear()` – очищает коллекцию от всех элементов.
- `map.size` – возвращает текущее количество элементов.

Для перебора коллекции `Map` есть 3 метода:

- `map.keys()` – возвращает итерируемый объект по ключам,
- `map.values()` – возвращает итерируемый объект по значениям,
- `map.entries()` – возвращает итерируемый объект по парам вида `[ключ, значение]`, этот вариант используется по умолчанию в `for..of`.

Кроме этого, `Map` имеет встроенный метод `forEach`, схожий со встроенным методом массивов `Array`

При создании `Map` мы можем указать массив (или другой итерируемый объект) с парами ключ-значение для инициализации,

Если у нас уже есть обычный объект, и мы хотели бы создать `Map` из него, то поможет встроенный метод Object.entries(obj), который получает объект и возвращает массив пар ключ-значение для него, как раз в этом формате. Так что мы можем создать `Map` из обычного объекта *`let map = new Map(Object.entries(obj));`*

Метод `Object.fromEntries`, который делает противоположное: получив массив пар вида `[ключ, значение]`, он создаёт из них объект

Объект `Set` – это особый вид коллекции: «множество» значений (без ключей), где каждое значение может появляться только один раз.

Его основные методы это:

- `new Set(iterable)` – создаёт `Set`, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый `Set`.
- `set.add(value)` – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект `set`.
- `set.delete(value)` – удаляет значение, возвращает `true`, если `value` было в множестве на момент вызова, иначе `false`.
- `set.has(value)` – возвращает `true`, если значение присутствует в множестве, иначе `false`.
- `set.clear()` – удаляет все имеющиеся значения.
- `set.size` – возвращает количество элементов в множестве.

Основная «изюминка» – это то, что при повторных вызовах `set.add()` с одним и тем же значением ничего не происходит, за счёт этого как раз и получается, что каждое значение появляется один раз.

`Set` имеет те же встроенные методы, что и `Map`:

- `set.values()` – возвращает перебираемый объект для значений,
- `set.keys()` – то же самое, что и `set.values()`, присутствует для обратной совместимости с `Map`,
- `set.entries()` – возвращает перебираемый объект для пар вида `[значение, значение]`, присутствует для обратной совместимости с `Map`.

Мы можем перебрать содержимое объекта set как с помощью метода `for..of`, так и используя `forEach`: Функция в `forEach` у `Set` имеет 3 аргумента: значение `value`, потом *снова то же самое значение* `valueAgain`, и только потом целевой объект. Это действительно так, значение появляется в списке аргументов дважды. Это сделано для совместимости с объектом `Map`, в котором колбэк `forEach` имеет 3 аргумента. Выглядит немного странно, но помогает легко в коде заменить `Map` на `Set` и наоборот.

1.5. Использование библиотек для создания клиентских приложений

Сайты на чистом JavaScript давно стали редкостью: обычно при разработке используют библиотеки и фреймворки. Одна из самых популярных библиотек — React. В этом разделе познакомимся с её основными принципами и идеями. Расскажем, что из себя представляет React, какие у него преимущества в сравнении с другими инструментами и как работать с React-компонентами.

1.5.1. Зачем нужны реактивные фреймворки и библиотеки?

Современные сайты отличаются от тех, которыми мы пользовались раньше: они более динамичны. С их помощью можно покупать товары, общаться с друзьями или писать комментарии.

Изменилась и структура сайтов. Изначально они состояли из набора отдельных HTML-страниц. Сейчас они представляют собой один HTML-документ, внешний вид которого меняется динамически через JavaScript.

Такой подход получил название SPA — single-page application, или одностраничное приложение. SPA делает сайты полноценными приложениями, которые не перезагружаются между переходами по страницам. Чтобы выполнить переход, нужно полностью перерисовать контент документа с помощью JavaScript и полученных данных.

Хотя требования к интерактивности сайтов выросли, инструменты для её создания почти не изменились.

решение задачи «в лоб» будет невероятно трудоёмким. Придётся написать тысячи строк кода взаимодействия с DOM API — такой код будет сложно читать и поддерживать. Использование библиотеки наподобие jQuery лишь поменяет интерфейс взаимодействия с DOM, но не решит проблему.

Можно написать свою реализацию MV* паттерна и настроить связывание данных, построив свою систему рендеринга. Но такие решения сложно реализовать быстро и качественно, а новым членам команды разработки будет непросто разобраться в чужих паттернах.

Подобные сложности привели к созданию реактивных фреймворков и библиотек. Идея реактивности такова: вместо того, чтобы каждый раз вручную в коде в императивном виде менять интерфейс приложения, гораздо удобнее изначально в декларативном виде задать связь между данными и их отображением и в дальнейшем в коде менять только данные. Интерфейс будет перерисовываться автоматически под новые данные.

То есть реактивный подход подразумевает работу с данными, при изменении которых автоматически меняются и другие части программы, в том числе интерфейс. React стал одной из первых и самой популярной на начало 2022 года реактивной библиотекой для работы с интерфейсами.

1.5.2. Что такое React?

React — это декларативная JavaScript-библиотека для создания пользовательских интерфейсов. Она позволяет собирать сложный UI из маленьких изолированных кусочков кода, называемых «компонентами».

- React предоставляет декларативное описание интерфейса — с ним удобнее работать, чем с вереницей императивных инструкций DOM API.
- React сам решает, какие узлы DOM-дерева в какой момент обновлять. Он не выполняет лишних операций, поэтому код на React быстрее, чем большинство наивных реализаций на чистом JavaScript.
- У React отличная документация. Самописные решения для реализации дата биндинга обычно плохо задокументированы или не задокументированы вовсе.
- Благодаря JSX в одном компоненте мы видим и логику, и разметку, с которой должна работать данная логика. Обычно они неразрывно связаны, и видеть их рядом намного удобнее, чем отдельно хранить HTML и JavaScript.
- Разбиение кода на компоненты позволяет переиспользовать код.

React предоставляет два синтаксиса для создания компонентов: классовый и функциональный. Классовый используется редко, так как синтаксис классов часто излишен для описания компонентов. Поэтому мы будем рассматривать функциональные компоненты.

Функциональный компонент в React — это JavaScript-функция, которая на входе получает объект с данными для отрисовки компонента — их называют props, а на выходе возвращает описание интерфейса, который нужно отобразить.

Пример с приветствием пользователя на React может выглядеть так:

```
import React from "react";
```

```
function HelloWorld (props) {
  return (<h1>Привет, Мир!!! Меня зовут <b>{props.name}</b></h1>);
}

export default HelloWorld;
```

В этом коде легко разобраться, даже если вы не знаете синтаксиса React. Здесь в виде функции описан React-компонент `HelloWorld`. На входе он получает объект с данными, props, который содержит свойство `name` с именем пользователя. Функция возвращает специальный код, похожий на HTML-разметку. Подробнее о нём расскажем позже, а сейчас лишь важно понимать, что этот код работает. React действительно отобразит на странице DOM-дерево — оно будет соответствовать разметке, которую вернуло `HelloWorld`. Но в отличие от обычного HTML, мы смогли подставить в разметку данные из объекта props `<b>{props.name}</b>`, тем самым шаблонизировав её.

Код на React выглядит легче и лаконичнее решения на чистом JS. Мы не описываем, как перерисовывать интерфейс. Вместо этого мы указываем, что нужно отобразить на месте компонента, и используем для этого синтаксис, похожий на HTML. Такой подход позволяет относительно легко решать и более сложные задачи. Например, перерисовать разметку страницы корзины и отобразить на её месте интерфейс страницы заказа.

Для своих перерисовок React использует Virtual DOM. Это позволяет обновлять только изменившиеся узлы DOM-дерева, а не перерисовывать всю страницу. Благодаря этому приложения не расходуют лишние ресурсы.

React решает проблемы фронтендеров при разработке интерфейсов динамичных сайтов и SPA-приложений. При использовании классической связки JavaScript и HTML перерисовка интерфейса сильно усложняется. React же не ограничивает возможности разработчика, в отличие от любых фреймворков.

1.5.3. Как работать с React-компонентами?

Рассмотрим подробнее синтаксис компонентов. Как мы уже говорили, React-компонент — это JavaScript-функция, которая на входе получает объект с данными для отрисовки компонента, а на выходе возвращает React-элемент.

React-элемент — это JavaScript-объект, описывающий узел DOM-дерева. Он имеет специальный формат, который React умеет обрабатывать и отображать на странице. Для создания таких объектов библиотека React предоставляет метод `React.createElement`.

Чтобы создать на странице разметку `<p>Тяжело читать <b>React.createElement</b>!!!</p>`, можно написать React-компонент:

```
import React from "react";

function HelloReact() {
  // не нужно думать, что это такое, дальше мы будем использовать
JSX
  return React.createElement(
    "p",
    {},
```

```
      "Тяжело читать ",
      React.createElement("b", {}, "React.createElement"),
      "!!!"
    );
}
```

Вызовы `React.createElement` не выглядят интуитивно понятными. Глядя на код, сложно понять, что будет выведено на страницу. Поэтому `React.createElement` обычно напрямую не применяется, а используется специальный синтаксис JSX.

1.5.4. Что такое JSX?

JSX — расширение JavaScript для описания интерфейса прямо в JS с помощью синтаксиса, похожего на HTML. С использованием JSX пример будет выглядеть так:

```
import React from "react";
function HelloReact() {
  return <p>Тяжело читать <b>React.createElement</b>!!!</p>
}
```

С JSX код стал понятнее. Сразу видно, какой фрагмент DOM-дерева описывает компонент.

JSX позволяет шаблонизировать разметку, подставляя в неё данные, вычисленные из JavaScript-выражений. Для этого используется синтаксис фигурных скобок. Если переписать пример из начала статьи на React, то он может выглядеть так:

```
import React from "react";

function HelloWorld (props) {
  return (<h1>Привет, Мир!!! Меня зовут <b>{props.name}</b></h1>);
}

export default HelloWorld;
```

Здесь компонент `HelloWorld` получает в объекте `props` имя пользователя `name` и на основе этих данных возвращает нужную разметку: `<h1>Привет, Мир!!! Меня зовут <b>{name}</b></h1>`. Благодаря синтаксису фигурных скобок значение из переменной `name` подставляется в JSX.

Однако браузер ничего не знает про JSX и не умеет его интерпретировать. При использовании JSX нужно настроить сборку проекта так, чтобы JSX автоматически трансформировался в JavaScript-объекты и в браузер попадал чистый JS. Для этого есть Babel-плагины и пресеты, а также проекты наподобие CreateReactApp. В статье мы используем онлайн IDE CodeSandbox, которая предоставляет готовый шаблон для React-проектов.

Теперь посмотрим, как заставить React вывести на экран JSX-разметку. Для этого React предоставляет метод `ReactDOM.render`. Первым параметром в него передаётся JSX-разметка. Вторым — узел DOM-дерева, в который нужно вставить кусочек интерфейса.

Чтобы вывести компонент `HelloWorld` в DOM-узел с id `root`, нужно написать:

```
import ReactDOM from "react-dom";
import HelloWorld from "./hello-world";

const rootElement = document.getElementById("root");
ReactDOM.render(HelloWorld({name:"Вася"}) , rootElement);
```

Вызов `HelloWorld` с объектом `{name:"Вася"}` вернёт нужную JSX-разметку, а `rootElement` хранит ссылку на DOM-элемент с id `root`. В результате вызов `ReactDOM.render(HelloWorld({name:"Вася"}) , rootElement)` вставит полученный из компонента интерфейс в нужный DOM-узел.

Вручную, как функции, компоненты обычно не вызывают. Дело в том, что в JSX можно вставлять компоненты как обычные HTML-теги, передавая данные через атрибуты. При отрисовке на месте компонента в JSX будет выведена та разметка, которую вернёт компонент.

Имя компонента в JSX должно начинаться с большой буквы, иначе JSX будет идентифицировать его как обычный HTML-тег.

Таким образом, вместо ReactDOM.render(HelloWorld({name:"Вася"}) , rootElement); пишут: ReactDOM.render(<HelloWorld name="Вася" /> , rootElement);.

1.5.5. Композиция компонентов и хуки

Возможность вставлять в JSX не только HTML-теги, но и свои компоненты — один из важнейших приёмов в React. Страницы сайта обычно бывают намного сложнее, чем пример `Hello World`: они состоят из сотен или даже тысяч HTML-элементов. Описывать такую страницу в одном модуле или компоненте — плохая идея. Код превратится в огромную простыню. Поэтому страницы разбивают на более мелкие кусочки интерфейса, и каждый из них реализуют в виде отдельного компонента. Этот приём в React называют **композицией компонентов**.

Тот же самый пример с `HelloWorld` можно представить в виде композиции двух маленьких компонентов:

```
// компонент Name
import React from "react";

function Name(props) {
  return <b>{props.name}</b>;
}
export default Name;

// компонент HelloWorld
import React from "react";
import Name from "./name";

function HelloWorld(props) {
  return (
    <h1>
```

```
        Привет, Мир! Меня зовут <Name name={props.name} />
      </h1>
    );
}
export default HelloWorld;
```

Теперь за отрисовку имени пользователя отвечает компонент `Name`. Его можно использовать не только в `HelloWorld`, но и любой другой части интерфейса, где необходимо вывести имя пользователя.

Для более сложных страниц композиция может быть больше

Таким образом, интерфейс в React представляет собой дерево компонентов, каждый из которых отвечает за свой кусочек интерфейса. Так как данные передаются только через props сверху вниз, от родительских компонентов к дочерним, это образует однонаправленный поток данных. Работа с ними становится более предсказуемой и понятной.

Создаваемые с помощью React пользовательские интерфейсы не статичны. Мы можем их перерисовать в ответ на действия пользователя. Для этого React предоставляет React-хуки.

Хуки — это функции, которые позволяют изменить стандартное поведение системы собственным кодом. При этом интерфейс функции ограничивает эти изменения безопасными для системы пределами.

Рекомендуем изучить статью от самого создателя React Дена Абрамова https://habr.com/ru/company/ruvds/blog/445276/

В случае с React хуки помогают управлять жизненным циклом компонента, позволяя в нужный момент вызвать, например, его перерисовку.

В результате счётчик на React может выглядеть так:

```
import React, { useState } from "react";

export default function Counter({ initValue }) {
  const [value, setValue] = useState(initValue || 1);

  return (
    <div>
      <button type="button" onClick={() => setValue(value - 1)}>-
</button>
      <span> {value} </span>
      <button type="button" onClick={() => setValue(value +
1)}>+</button>
    </div>
  );
}
```

`useState` — специальный React-хук для хранения состояния компонента. Когда состояние меняется, компонент перерисовывается.

При вызове `useState` возвращает массив, первый элемент которого — значение состояния. Во втором элементе хранится функция-сеттер, с помощью которой можно менять значение состояния. Для кнопок передан props `onClick`. В него записана функция, которая вызывается при нажатии на кнопку — `setValue` с новым значением состояния. Это вызывает перерисовку, и на экране отображается актуальное значение `value`.

`useState` — только один из множества доступных хуков. Более того, при необходимости разработчик может писать свои кастомные хуки.

1.5.6. Жизненный цикл компоненты и оптимизация рендеринга

Функциональные компоненты в React не имеют собственного жизненного цикла, как классовые (`componentDidMount`, `componentDidUpdate`, `componentWillUnmount`), но используя React Hooks мы можем управлять поведением компонента на разных этапах его жизни.

Этап	Хук	Описание
Монтирование	`useEffect(() => {}, [])`	Выполняется один раз после первого рендера
Обновление	`useEffect(() => {}, [deps])`	Выполняется при изменении зависимостей `deps`
Размонтирование	`useEffect(() => { return () => {} }, [])`	Функция, которая записана в виде возвращаемого значения, вызывается при удалении компонента

Примеры

Монтирование компонента

Это момент, когда компонент *впервые* появляется в DOM.

Используется хук `useEffect` с пустым массивом зависимостей:

```
useEffect(() => {
  console.log("Компонент смонтирован");
}, []);
```

Применение: загрузка данных, подписка на события, инициализация.

Обновление компонента

Происходит при изменении состояния (`useState`) или пропсов.

Используется `useEffect` с массивом зависимостей:

```
useEffect(() => {
  console.log("Обновление компонента");
}, [someValue]);
```

Применение: реагирование на изменение данных, повторная загрузка, пересчёт.

Размонтирование компонента

Когда компонент удаляется из DOM.

Используется функция очистки внутри `useEffect`:

```
useEffect(() => {
  return () => {
    console.log("Компонент размонтирован");
  };
}, []);
```

Применение: отмена подписок, очистка таймеров, освобождение ресурсов.

Оптимизация рендеринга

Рендеринг — это процесс, при котором React обновляет виртуальный DOM и, при необходимости, реальный DOM. Избыточные рендеры могут замедлить работу приложения.

Основные техники оптимизации

1. React.memo

Позволяет избежать повторного рендера компонента, если его пропсы не изменились.

```
const MyMemoComponent = React.memo(myComponent);
```

2. useMemo

Мемоизирует результат вычислений, чтобы заново не пересчитывать их при каждом рендере. **Не злоупотребляйте мемоизацией** — она полезна при дорогих вычислениях, но может усложнить код.

```
const expensiveResult = useMemo(() => {
  return computeHeavy(data);
}, [data]);
```

3. useCallback

Мемоизирует функцию, чтобы не создавать её заново при каждом рендере.

```
const handleClick = useCallback(() => {
  doSomething();
}, []);
```

Также стоит помнить про стандартные советы:

- Старайтесь использовать ключи (key) при рендере списков, чтобы React мог эффективно отслеживать изменения.
- Разделение компонентов. Разделяйте большие компоненты на более мелкие, чтобы уменьшить область рендера.
- Избегайте анонимных функций и объектов в пропсах. Ведь каждый раз при рендере создаются новые ссылки, что может вызвать лишние обновления.
- Используйте инструменты профилирования — React DevTools поможет выявить узкие места.

Типичные ошибки

Отсутствие зависимостей в useEffect — может привести к бесконечному циклу рендеров. Если забыть записать в useEffect второй параметр, то функция будет вызываться при каждом ререндере.

Неправильное использование React.memo — не работает, если передаются в компонент в качестве пропсов новые объекты или новые функции.

```
<MyButton onClick={() => doSomething()} />
```

При каждом рендере создаётся новая анонимная функция () => doSomething(), что вызовет лишние рендеры дочернего компонента MyButton.

Правильно делать так:

```
const handleClick = useCallback(() => doSomething(), []);
<MyButton onClick={handleClick} />
```

Мемоизация без необходимости — useMemo и useCallback не нужны, если вычисления лёгкие.

Отсутствие очистки в useEffect. Подписки на какие-то события или таймеры продолжат работать даже после размонтирования компонента.

```
useEffect(() => {
  const timer = setInterval(doSomething, 1000);
}, []);
```

Правильно так:

```
useEffect(() => {
  const timer = setInterval(doSomething, 1000);
  return () => clearInterval(timer);
}, []);
```

Приведем примеры кода с ошибками и покажем как их исправить.

Пример ошибки: fetch вне useEffect.

```
import { useState } from 'react';

function BrokenFetch() {
  const [data, setData] = useState(null);

  // fetch вызывается прямо в теле компонента
  fetch('https://jsonplaceholder.typicode.com/posts/1')
    .then(res => res.json())
    .then(json => setData(json));

  return (
    <div>{data ? <p>{data.title}</p> : <p>Загрузка...</p>}</div>
  );
}
```

Почему это ошибка? Запрос fetch вызывается при каждом рендере, потому что он находится в теле компонента, а не внутри хука useEffect.

setData обновляет состояние → вызывает рендер → снова fetch → снова setData → 🔁 бесконечный цикл.

- Всегда размещайте побочные эффекты (запросы, таймеры, подписки) внутри useEffect.
- Добавляйте массив зависимостей, чтобы контролировать, когда эффект должен сработать.
- Не вызывайте setState в теле компонента — только внутри событий или эффектов.

Правильный вариант с useEffect

```
import { useState, useEffect } from 'react';

function FixedFetch() {
  const [data, setData] = useState(null);
```

```
  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/posts/1')
      .then(res => res.json())
      .then(json => setData(json));
  }, []); // вызывается только один раз при монтировании

  return (
    <div> {data ? <p>{data.title}</p> : <p>Загрузка...</p>} </div>
  );
}
```

```
import React, { useState, useEffect, useMemo } from 'react';
function Demo({ data }) {
  useEffect(() => {    fetchData();  });
  const [value, setValue] = useState(0);
  if (value > 5) {    const [extra, setExtra] = useState(10);  }
  const result = useMemo(() => {    return 2 + 2;  }, [value]);
  const handleClick = () => {    console.log("Clicked");  };

  return (
    <div>
      <button onClick={() => handleClick()}>Click</button>
      <p>{result}</p>
    </div>
  );
}
```

Ошибки в приведенном выше коде:

Ошибка	Объяснение
`useEffect` без массива зависимостей	Будет вызываться при каждом рендере
`useState` внутри условия	Хуки нельзя вызывать внутри условий
`useMemo` для простого выражения	Мемоизация не нужна для `2 + 2`
Анонимная функция в `onClick`	Создаётся новая функция при каждом рендере
`fetchData()` не определена	Нет импорта или определения функции
Нет `key` при рендере списка (если бы был список)	Нарушение правил рендера коллекций
Нет проверки `data` перед использованием	Может быть `undefined` или `null`

```
import React, { useState, useEffect, useCallback } from 'react';

// Предположим, что fetchData определена где-то выше
import { fetchData } from './api';
```

```
function Demo({ data = [] }) {
  const [value, setValue] = useState(0);
  const [extra, setExtra] = useState(10);

  useEffect(() => {
    fetchData();
  }, []);

  const handleClick = useCallback(() => {
    console.log("Clicked");
  }, []);

  return (
    <div>
      <button onClick={handleClick}>Click</button>
      <p>Счётчик: {value}</p>
      <p>Доп. значение: {extra}</p>
      <ul>
        {data.map((item, index) => (
          <li key={index}>{item}</li>  {/* добавлен key */}
        ))}
      </ul>
    </div>
  );
}
```

Пример компонента, который должен загружать данные при изменении userId. Однако он вызывает бесконечный рендер. В чем ошибка? Нельзя вызывать setState без условий внутри useEffect, реагирующего на state.

```
import { useState, useEffect } from 'react';

function UserLoader({ userId }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetch(`https://api.example.com/users/${userId}`)
      .then(res => res.json())
      .then(data => setUser(data));
  });

  return (
    <div>
      {user ? <p>{user.name}</p> : <p>Загрузка...</p>}
    </div>
  );
}
```

Хук useEffect вызывается при каждом рендере, потому что не указан массив зависимостей. setUser обновляет состояние → вызывает рендер → снова useEffect → снова setUser → 🔁 бесконечный цикл.

Исправим

```
import { useState, useEffect } from 'react';

function UserLoader({ userId }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetch(`https://api.example.com/users/${userId}`)
      .then(res => res.json())
      .then(data => setUser(data));
  }, [userId]); // добавлена зависимость

  return (
    <div>
      {user ? <p>{user.name}</p> : <p>Загрузка...</p>}
    </div>
  );
}
```

1.5.7. Роутинг. Создание SPA

В одностраничных приложениях (SPA) на React нет традиционных переходов между HTML-страницами. Вместо этого используется роутинг, который позволяет отображать разные компоненты в зависимости от URL, не перезагружая страницу.

Обычно используют библиотеки для роутинга. Наиболее популярная библиотека — React Router.

Основные компоненты React Router

Компонент	Назначение
BrowserRouter	Оборачивает всё приложение, включает поддержку истории
Routes	Контейнер для всех маршрутов
Route	Определяет путь и компонент, который нужно отобразить
Link	Создаёт ссылку без перезагрузки страницы
useNavigate	Хук для программной навигации
useParams	Хук для получения параметров из URL

Пример базовой настройки роутинга

```
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';
import Home from './Home';
import About from './About';

function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Главная</Link>
        <Link to="/about">О нас</Link>
      </nav>
```

```
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
}
```

Как добавляются динамические маршруты.

```
<Route path="/user/:id" element={<UserProfile />} />
```

Внутри компоненты UserProfile можно получить id так:

```
import { useParams } from 'react-router-dom';

function UserProfile() {
  const { id } = useParams();
  return <p>Профиль пользователя: {id}</p>;
}
```

Реализация программной навигации

```
import { useNavigate } from 'react-router-dom';

function LoginButton() {
  const navigate = useNavigate();

  const handleLogin = () => {
    // логика авторизации
    navigate('/dashboard');
  };

  return <button onClick={handleLogin}>Войти</button>;
}
```

Советы

Всегда оборачивайте приложение в BrowserRouter.

Разделяйте маршруты по логическим модулям — это улучшает читаемость и масштабируемость.

Используйте Link вместо <a> — это предотвращает перезагрузку страницы.

Не забывайте про useParams и useNavigate — они упрощают работу с маршрутом.

Защищенный роутинг

Настройка защищённых маршрутов (protected routes) в React позволяет ограничить доступ к определённым страницам только для авторизованных пользователей. Это особенно важно для панелей администратора, личных кабинетов и других приватных разделов приложения.

Что такое защищённый маршрут? Это маршрут, который отображает компонент только если пользователь авторизован. В противном случае — перенаправляет, например, на страницу входа.

```
// AuthContext.js Создаём контекст авторизации
import { createContext, useContext, useState } from 'react';

const AuthContext = createContext(null);
```

```
export function AuthProvider({ children }) {
  const [user, setUser] = useState(null);

  const login = (userData) => setUser(userData);
  const logout = () => setUser(null);

  return (
    <AuthContext.Provider value={{ user, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
}

export function useAuth() {
  return useContext(AuthContext);
}
```

И компонент ProtectedRoute

```
// ProtectedRoute.js
import { Navigate } from 'react-router-dom';
import { useAuth } from './AuthContext';

export default function ProtectedRoute({ children }) {
  const { user } = useAuth();

  if (!user) {    return <Navigate to="/login" replace />; }

  return children;
}
```

Теперь можно использовать ProtectedRoute в маршрутах.

```
// App.js
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import { AuthProvider } from './AuthContext';
import ProtectedRoute from './ProtectedRoute';

import Home from './pages/Home';
import Login from './pages/Login';
import Dashboard from './pages/Dashboard';

function App() {
  return (
    <AuthProvider>
      <BrowserRouter>
        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/login" element={<Login />} />
          <Route
            path="/dashboard"
            element={
              <ProtectedRoute>
                <Dashboard />
              </ProtectedRoute>
```

```
            }
          />
        </Routes>
      </BrowserRouter>
    </AuthProvider>
  );
}
```

Пример страницы регистрации

```
// Login.js
import { useAuth } from './AuthContext';
import { useNavigate } from 'react-router-dom';

function Login() {
  const { login } = useAuth();
  const navigate = useNavigate();

  const handleLogin = () => { login({ name: 'Student' });
    navigate('/dashboard');
  };

  return <button onClick={handleLogin}>Войти</button>;
}
```

1.5.8. Управление состоянием в React

Состояние — это данные, которые определяют поведение и отображение компонентов. Грамотное управление состоянием делает приложение предсказуемым, масштабируемым и удобным для поддержки. Это фундамент для создания интерактивных интерфейсов. В неё входят:

Локальное состояние (`useState`, `useReducer`). Подходит для простых компонентов

- Глобальное состояние. Контекст: `React.createContext`, `useContext`. Подходит для передачи данных между компонентами без пропсов
- Асинхронное состояние. `useEffect`, `useAsync`, `React Query` Загрузка данных, обработка ошибок.
- Внешние библиотеки (Redux, Zustand, Recoil, Jotai). Используются в крупных приложениях

Локальное состояние

Используется внутри одного компонента. Подходит для простых задач: формы, переключатели, счётчики.

useState

Изменение состояния вызывает повторный рендер. Можно использовать для хранения строк, чисел, объектов, массивов.

useReducer

Подходит для более сложной логики, особенно при множестве действий.

```
const reducer = (state, action) => {
```

```
  switch (action.type) {
    case 'increment': return { count: state.count + 1 };
    case 'decrement': return { count: state.count - 1 };
    default: return state;
  }
};
const [state, dispatch] = useReducer(reducer, { count: 0 });
```

Глобальное состояние

Используется для передачи данных между компонентами, которые не связаны напрямую.

React Context

Создаёт хранилище, доступное из любого компонента. Подходит для чего-то “глобального”: тема (светлая/тёмная), язык, авторизации. Не оптимален для часто меняющихся данных.

```
const ThemeContext = createContext();

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar() {
  const theme = useContext(ThemeContext);
  return <div className={theme}>...</div>;
}
```

Асинхронное состояние

Используется для загрузки данных с сервера, обработки ошибок, отображения загрузки. Асинхронное состояние — это данные, которые приходят не сразу, а с задержкой: из API, базы данных, локального хранилища и т. д. Управление таким состоянием требует особого подхода, чтобы избежать ошибок, утечек памяти и бесконечных рендеров.

useEffect + fetch

```
useEffect(() => {
  fetch('/api/data')
    .then(res => res.json())
    .then(setData)
    .catch(setError);
}, []);
```

Есть библиотеки, которые упрощают работу с асинхронными запросами (кэширование, повторные запросы, управление загрузкой и ошибками):

Библиотека	Особенности
React Query	Кэширование, повторные запросы, автоматическое обновление
SWR	Простота, автообновление, кэш
Axios	Удобный клиент для HTTP-запросов
TanStack Query	Расширенная версия React Query с поддержкой серверного рендеринга

Пример базовой загрузки

```
import { useState, useEffect } from 'react';

function UserList() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch('https://jsonplaceholder.typicode.com/users')
      .then(res => {
        if (!res.ok) throw new Error('Ошибка загрузки');
        return res.json();
      })
      .then(data => setUsers(data))
      .catch(err => setError(err.message))
      .finally(() => setLoading(false));
  }, []);

  if (loading) return <p>Загрузка...</p>;
  if (error) return <p>Ошибка: {error}</p>;

  return (
    <ul>
      {users.map(user => <li key={user.id}>{user.name}</li>)}
    </ul>
  );
}
```

Важные принципы

Асинхронность нельзя использовать напрямую в useEffect:

```
// Нельзя:
useEffect(async () => { ... });

// Нужно:
useEffect(() => {
  async function loadData() {
    const res = await fetch(...);
  }
  loadData();
}, []);
```

Обрабатывайте ошибки — всегда добавляйте .catch() или try/catch.

Добавляйте индикаторы загрузки — это улучшает UX.

Очищайте эффекты, если компонент может быть размонтирован до завершения запроса.

Очистка асинхронных эффектов

```
useEffect(() => {
  const controller = new AbortController();

  fetch('/api/data', { signal: controller.signal })
    .then(res => res.json())
    .then(setData)
    .catch(err => {
      if (err.name !== 'AbortError') {
        setError(err.message);
      }
    });

  return () => controller.abort(); // отмена запроса при
размонтировании
}, []);
```

Использование async/await

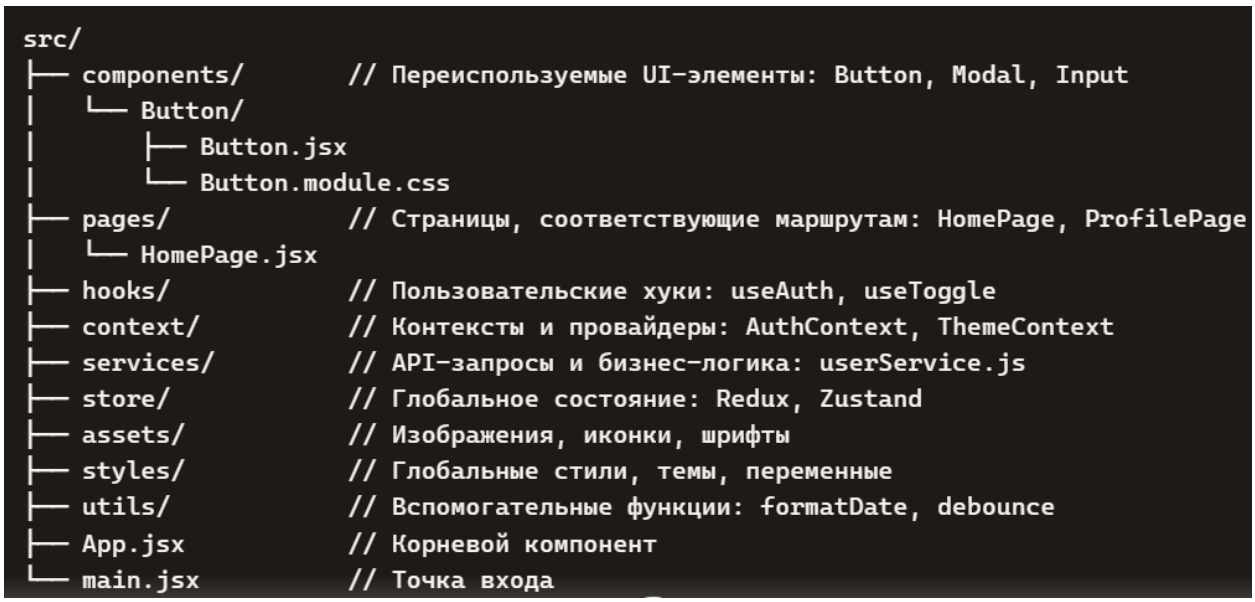
```
useEffect(() => {
  const loadData = async () => {
    try {
      const res = await fetch('/api/data');
      const json = await res.json();
      setData(json);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  };

  loadData();
}, []);
```

1.5.9. Архитектура React-приложения

Хорошо организованная структура проекта — это основа читаемого и масштабируемого кода. Она помогает быстро находить нужные файлы, разделять ответственность и упрощает командную работу. Надо разделять по компоненты назначению: UI, логика, данные. Не смешивать стили, логику и JSX в одном файле без необходимости. Каждый компонент — в своей папке с логикой, стилями и тестами.

Для не очень сложных учебных проектов рекомендуется следующая структура



Типы компонентов

Разделение компонентов по типу помогает соблюдать принцип SRP (Single Responsibility Principle) — ключ к читаемому и переиспользуемому коду и упрощает тестирование.

Презентационные компоненты (UI)

Отвечают только за отображение. UI-компоненты должны быть максимально «глупыми».

Получают данные через пропсы.

Не содержат состояния или бизнес-логики.

Пример:

```
function UserCard({ name, avatar }) {
  return (
    <div className="card">
      <img src={avatar} alt={name} />
      <p>{name}</p>
    </div>
  );
}
```

Контейнерные компоненты (“умные” компоненты)

Контейнеры — точка входа для логики и данных. Отвечают за загрузку данных, управление состоянием, вызов API.

Используют хуки, контексты, сервисы. Передают данные в UI-компоненты.

Пример:

```
function UserContainer() {
  const [user, setUser] = useState(null);

  useEffect(() => {
    getUser().then(setUser);
```

```
  }, []);

  return user ? <UserCard {...user} /> : <p>Загрузка...</p>;
}
```

Управление состоянием

Состояние — это данные, которые определяют поведение и отображение компонентов. В React принято разделять состояние на три уровня:

Локальное состояние

Используется внутри одного компонента. Подходит для UI-состояний: модальные окна, чекбоксы, поля ввода.

Глобальное состояние

Используется между несколькими компонентами. Подходит для авторизации, темы, языка, корзины.

Используй специальные инструменты для хранения: Context API — для простых случаев, Redux, Zustand, Jotai — для сложных приложений.

Не храни всё в Redux — локальное состояние должно оставаться локальным.

Серверное состояние

Данные, полученные из внешних или внутренних API. Подходит для списков, профилей, статистики.

Можно использовать специальные инструменты: React Query, RTK Query, SWR

Рекомендация: не смешивай UI-состояние и серверные данные в одном хранилище. Разделяй UI-состояние и бизнес-логику. Используй `React Query` для загрузки и кэширования данных.

Работа с API

Запросы к серверу — это побочные эффекты, которые должны быть изолированы от компонентов. Соблюдай просто е правило: «все запросы — в сервисах»

Рекомендация:

- Не размещай fetch прямо в JSX.
- Обрабатывай ошибки и загрузку.
- Используй AbortController для отмены запроса при размонтировании.

Пример:

```
// services/userService.js
export const getUser = (id) =>
  fetch(`/api/users/${id}`).then((res) => res.json());
Использование в компоненте:
import { getUser } from '../services/userService';

useEffect(() => {
  getUser(userId).then(setUser);
}, [userId]);
```

1.5.10. Redux Toolkit в React: управление состоянием

Redux — это библиотека для управления состоянием JavaScript-приложений. Была создана Дэном Абрамовым в 2015 году. Она была разработана, чтобы решить проблемы, связанные с управлением состоянием в сложных приложениях. Redux используется в различных технологиях, включая React, Angular, Vue и другие, и является популярной и влиятельной библиотекой. Она особенно полезна тогда в React, когда данные должны быть доступны в разных частях интерфейса, а логика их изменения должна быть централизованной и предсказуемой.

Конечно React сразу предоставляет локальное состояние (useState) и контекст (Context API), но в сложных приложениях этого становится недостаточно. Redux решает несколько ключевых проблем:

- Централизация состояния: все данные хранятся в одном месте.
- Предсказуемость: изменения происходят только через строго определённые действия.
- Удобство отладки: можно отслеживать историю изменений через DevTools.
- Масштабируемость: легко добавлять новые модули и логику.

Redux строится на трёх фундаментальных принципах:

- Единое хранилище (store) Всё состояние приложения хранится в одном объекте.
- Изменения через actions Состояние нельзя менять напрямую — только через специальные объекты-действия.
- Чистые функции (reducers) Reducer — это функция, которая получает текущее состояние и action, и возвращает новое состояние без побочных эффектов.

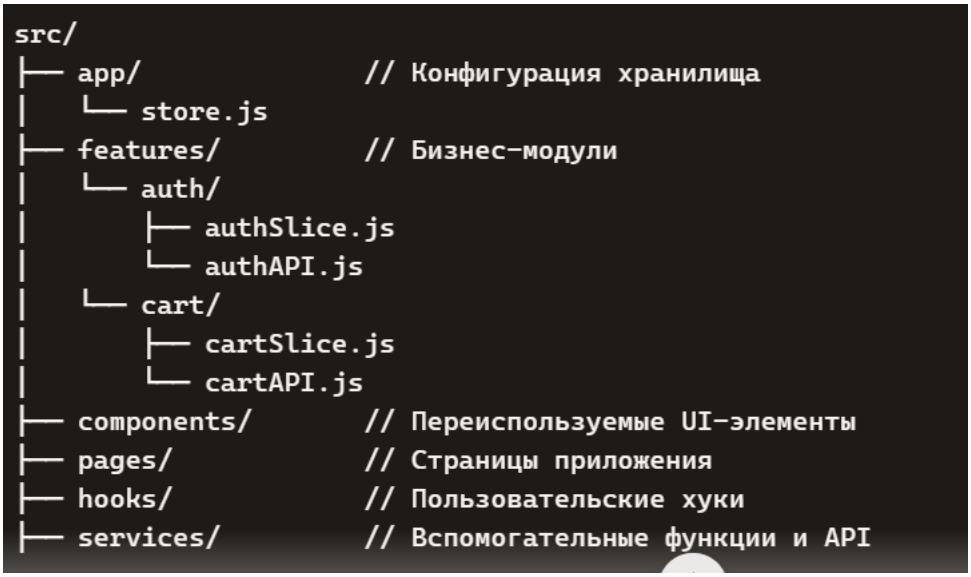
Ключевые элементы Redux

Элемент	**Назначение**
store	Центральное хранилище состояния
action	Объект с типом действия и (опционально) данными
reducer	Функция, описывающая, как состояние меняется в ответ на action
dispatch	Метод, запускающий action и передающий его в reducer
useSelector	Хук для получения данных из store
useDispatch	Хук для отправки действий

Redux Toolkit — это современный способ работы с глобальным состоянием в React. Он даёт набор инструментов и упрощает архитектуру, устраняет шаблонный код и предоставляет мощные инструменты для синхронной и асинхронной логики. В отличие от подхода с ручным созданием **store, actions и reducers**, Redux Toolkit объединяет всё это в единый, удобный API.

Благодаря встроенной библиотеке Immer, можно писать мутабельный код (`state.items.push(...)`), который автоматически преобразуется в иммутабельный.

Рекомендуемая структура проекта



Каждый модуль (features) содержит slice и API-запросы

Хранилище (store.js) объединяет все редьюсеры

Компоненты используют useSelector и useDispatch для доступа к данным

Основные концепции Redux Toolkit

configureStore

Функция для создания хранилища приложения. Она объединяет все срезы (slice) и автоматически подключает DevTools и middleware.

```
import { configureStore } from '@reduxjs/toolkit';
import authReducer from './authSlice';
import cartReducer from './cartSlice';

export const store = configureStore({
  reducer: {
    auth: authReducer,
    cart: cartReducer,
  },
});
```

createSlice

Slice — это модуль, объединяющий состояние, действия и редьюсер в одном месте.

Функция ***createSlice*** объединяет: начальное состояние initialState, список действий reducers, редьюсер, который обрабатывает эти действия.

Каждый slice обычно описывает одну бизнес-сущность: пользователь, корзина, задачи и т. д.

До появления Redux Toolkit разработчики вручную создавали (рисунок 74):

- action-creator'ы

- типы действий
- редьюсеры с switch-case
- отдельные файлы для каждого элемента

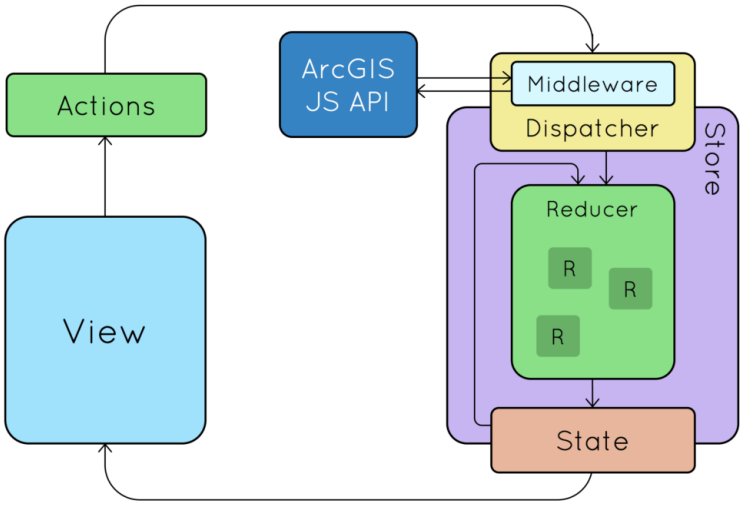


Рисунок 74. Схема работы Redux

Использование createSlice устраняет этот повторяющийся шаблонный код.

Синтаксис createSlice

Аргументы:

name — уникальное имя среза, используется для генерации типа действия, initialState — объект, описывающий начальное состояние, reducers — объект с функциями, каждая из которых описывает одно действие

```
import { createSlice } from '@reduxjs/toolkit';

const slice = createSlice({
  name: 'имяСреза',
  initialState: {}, // начальное состояние
  reducers: {
    действие1: (state, action) => { ... },
    действие2: (state, action) => { ... },
  },
});
```

Пример: корзина товаров

```
// features/cart/cartSlice.js
import { createSlice } from '@reduxjs/toolkit';

const cartSlice = createSlice({
  name: 'cart',
  initialState: {
    items: [],
    total: 0,
  },
  reducers: {
    addItem: (state, action) => {
      state.items.push(action.payload);
      state.total += action.payload.price;
    },
    removeItem: (state, action) => {
```

```
          const index = state.items.findIndex(item => item.id === ac-
tion.payload);
          if (index !== -1) {
            state.total -= state.items[index].price;
            state.items.splice(index, 1);
          }
        },
        clearCart: (state) => {
          state.items = [];
          state.total = 0;
        },
      },
    });

    export const { addItem, removeItem, clearCart } = cart-
Slice.actions;
    export default cartSlice.reducer;
```

Использование в компоненте

```
import { useSelector, useDispatch } from 'react-redux';
import { addItem, removeItem } from '../features/cart/cartSlice';

function CartControls({ product }) {
  const items = useSelector(state => state.cart.items);
  const dispatch = useDispatch();

  return (
    <div>
      <button onClick={() =>
dispatch(addItem(product))}>Добавить</button>
      <button onClick={() =>
dispatch(removeItem(product.id))}>Удалить</button>
      <p>Всего товаров: {items.length}</p>
    </div>
  );
}
```

useSelector и useDispatch

В Redux Toolkit для взаимодействия компонентов с хранилищем используются два ключевых хука: useSelector — доступ к данным из хранилища (позволяет компоненту «подписаться» на нужный фрагмент состояния из Redux-хранилища.), useDispatch — отправка действий (позволяет компоненту инициировать изменение состояния, отправив действие (action) в хранилище).

Не нужно передавать `dispatch` через пропсы — каждый компонент может получить его сам. Сам dispatch принимает action, созданный через createSlice.

```
import { useSelector, useDispatch } from 'react-redux';
import { addItem } from '../store/cartSlice';

function CartButton({ product }) {
  const items = useSelector(state => state.cart.items);
  const dispatch = useDispatch();
```

```
  return (
    <button onClick={() => dispatch(addItem(product))}>
      Добавить в корзину ({items.length})
    </button>
  );
}
```

createAsyncThunk

Функция для описания асинхронных действий: загрузка данных, авторизация, отправка форм. Она автоматически создаёт три состояния: pending, fulfilled, rejected.

```
import { createAsyncThunk } from '@reduxjs/toolkit';

export const fetchUser = createAsyncThunk('auth/fetchUser', async
(id) => {
  const response = await fetch(`/api/users/${id}`);
  return await response.json();
});
```

Используется внутри extraReducers:

```
const authSlice = createSlice({
  name: 'auth',
  initialState: { user: null, loading: false },
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(fetchUser.pending, (state) => {
        state.loading = true;
      })
      .addCase(fetchUser.fulfilled, (state, action) => {
        state.user = action.payload;
        state.loading = false;
      })
      .addCase(fetchUser.rejected, (state) => {
        state.loading = false;
      });
  },
});
```

1.5.11. Введение в Next.js – фреймворк для React- разработки

Next.js — это мощный фреймворк поверх React, и про него стоит поговорить сразу после рассмотрения базовых концепций React.

Next.js — это фреймворк на базе React, разработанный компанией Vercel. Он предоставляет мощные инструменты для создания производительных, масштабируемых и SEO-дружественных веб-приложений. В отличие от чистого React, который отвечает только за отображение интерфейса, Next.js предлагает готовую инфраструктуру для серверного рендеринга, маршрутизации, работы с API и многого другого.

Что программист получает сразу “из коробки”:

- Серверный рендеринг (SSR) — позволяет генерировать HTML на сервере, улучшая SEO и ускоряя загрузку страниц.
- Статическая генерация (SSG) — создаёт HTML во время сборки, обеспечивая молниеносную загрузку.
- Гибкая маршрутизация — маршруты создаются автоматически на основе структуры файлов в папке pages/.
- API-роуты — возможность писать серверный код прямо в проекте.
- Оптимизация изображений — встроенный компонент next/image автоматически оптимизирует изображения.
- Поддержка модных технологий TypeScript, CSS-модулей, Sass, Tailwind CSS.
- Интеграция с Vercel — простое и быстрое развёртывание проекта.

Установка и запуск проекта

```
npx create-next-app@latest my-next-app
```

После этого запуска приложение будет развернуто в следующую структуру.

```
my-next-app/
├── pages/          # Страницы и маршруты
│   ├── index.js    # Главная страница (/)
│   └── about.js    # Страница "О нас" (/about)
├── public/         # Статические файлы (изображения, иконки и т.д.)
├── styles/         # CSS и SCSS файлы
├── components/     # Повторно используемые компоненты
├── next.config.js  # Конфигурация Next.js
└── package.json    # Зависимости и скрипты
```

Маршрутизация в Next.js

Общие принципы

В Next.js маршруты создаются автоматически на основе структуры файлов в папке pages/. Это называется файловая маршрутизация. Каждый .js, .jsx, .ts, или .tsx файл в pages/ становится отдельной страницей, доступной по соответствующему URL.

```
pages/
├── index.js      → /
├── about.js      → /about
├── contact.js    → /contact
├── blog/
│   └── post.js   → /blog/post
```

Статические маршруты

Это обычные страницы, соответствующие фиксированным путям.

```
// pages/about.js
export default function About() {
  return <h1>О нас</h1>;
}
```

Доступ по адресу: /about

Динамические маршруты

Next.js позволяет создавать маршруты с переменными, используя квадратные скобки в имени файла.

```
// pages/blog/[slug].js
import { useRouter } from 'next/router';

export default function BlogPost() {
  const router = useRouter();
  const { slug } = router.query;

  return <h1>Пост: {slug}</h1>;
}
```

Если пользователь перейдёт по адресу /blog/react-hooks, то slug будет равен "react-hooks".

Вложенные маршруты

Можно создавать вложенные папки в pages/, чтобы организовать структуру сайта.

Пример:

```
pages/
├── dashboard/
│   ├── index.js      → /dashboard
│   └── settings.js   → /dashboard/settings
```

Навигация между страницами

Для перехода между страницами используется компонент Link из next/link.

```
import Link from 'next/link';

export default function Home() {
  return (
    <div>
      <h1>Главная</h1>
      <Link href="/about">О нас</Link>
    </div>
  );
}
```

Программная навигация

Можно переходить между страницами программно с помощью хука useRouter.

```
import { useRouter } from 'next/router';

export default function GoToContact() {
  const router = useRouter();

  const handleClick = () => {
    router.push('/contact');
  };
```

```
  return <button onClick={handleClick}>Связаться с нами</button>;
}
```

Комбинированные маршруты

Можно комбинировать динамические и вложенные маршруты:

```
pages/
├── blog/
│   └── [category]/
│       └── [slug].js → /blog/javascript/react-hooks
```

Обработка 404 и ошибок

Next.js позволяет создать собственную страницу ошибки 404:

```
// pages/404.js
export default function Custom404() {
  return <h1>Страница не найдена</h1>;
}
```

Также можно создать страницу для обработки ошибок:

```
js
// pages/_error.js
function Error({ statusCode }) {
  return (
    <p>
      {statusCode
        ? `Ошибка ${statusCode}`
        : 'Произошла ошибка на клиенте'}
    </p>
  );
}

Error.getInitialProps = ({ res, err }) => {
  const statusCode = res ? res.statusCode : err ? err.statusCode :
404;
  return { statusCode };
};

export default Error;
```

Роутинг с параметрами и query

Next.js позволяет получать параметры из URL через useRouter:

```
// pages/search.js
import { useRouter } from 'next/router';

export default function SearchPage() {
  const router = useRouter();
  const { q } = router.query;

  return <h1>Результаты поиска: {q}</h1>;
}
```

URL: /search?q=nextjs → отобразится Результаты поиска: nextjs

1.5.12. Middleware и редиректы

С версии Next.js 12+ можно использовать middleware для обработки запросов на уровне маршрутов — например, для авторизации, редиректов, логирования.

```
// middleware.js
import { NextResponse } from 'next/server';

export function middleware(request) {
  const url = request.nextUrl.clone();
  if (!request.cookies.get('auth')) {
    url.pathname = '/login';
    return NextResponse.redirect(url);
  }
  return NextResponse.next();
}
```

1.5.13. Получение данных

Next.js предлагает три способа получения данных:

- `getStaticProps` — статическая генерация;
- `getServerSideProps` — серверный рендеринг;
- `getStaticPaths` — генерация путей для динамических страниц.

Это не просто имена — это **специальные методы**, которые Next.js распознаёт и использует по-разному при рендеринге страницы. При сборке проекта ищутся эти конкретные функции в файлах страниц (pages/*.js) и Next.js решает, как рендерить страницу:

Имя функции	**Когда вызывается**	**Как влияет на рендеринг**
`getStaticProps`	Один раз при сборке (npm run build)	Страница становится статически сгенерированной
`getServerSideProps`	При каждом запросе пользователя	Страница рендерится на сервере в реальном времени

Если назвать функцию по-другому — например, getProps или fetchData — Next.js её просто проигнорирует. Она не будет участвовать в рендеринге, и данные не попадут в props.

Как это работает внутри?

Оба метода — getStaticProps и getServerSideProps — используются в Next.js для получения данных до рендеринга страницы. Но они работают по-разному и применяются в разных ситуациях.

Как выбрать?

Если данные редко меняются → getStaticProps

Если данные зависят от запроса или часто обновляются → getServerSideProps.

getStaticProps — Статическая генерация

Что делает:

Получает данные на этапе сборки проекта (build time).

Генерирует HTML один раз, и он не меняется при каждом запросе.

Быстро загружается, идеально для статичных страниц, которые редко меняются.

Когда использовать:

Контент не зависит от запроса пользователя.

Данные обновляются редко (например, раз в день).

Примеры: блог, страница «О нас», список продуктов.

Пример:

```
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/posts');
  const posts = await res.json();

  return {
    props: { posts },
  };
}
```

getServerSideProps — Серверный рендеринг

Что делает:

Получает данные при каждом запросе к странице.

HTML генерируется на сервере “ на лету” – в реальном времени.

Подходит для страниц с динамическими данными, которые часто меняются.

Когда использовать:

Данные зависят от запроса (например, авторизация, параметры URL).

Нужно показывать актуальную информацию.

Примеры: личный кабинет, панель администратора, результаты поиска.

Пример:

```
export async function getServerSideProps(context) {
  const res = await
fetch(`https://api.example.com/user/${context.params.id}`);
  const user = await res.json();

  return {
    props: { user },
```

```
  };
}
```

Сравнение

Характеристика	**getStaticProps**	**getServerSideProps**
Когда вызывается	При сборке проекта (build time)	При каждом запросе к странице
Производительность	Очень высокая	Зависит от скорости сервера
SEO	Отличная	Отличная
Доступ к параметрам запроса	Нет	Да (context.query, context.params)
Подходит для	Статичных страниц	Динамических страниц
Примеры	Блог, каталог	Профиль пользователя, поиск

Сравнение на одном примере

Создадим страницу, которая показывает список новостей с внешнего API.

статическая генерация **`getStaticProps`**/ серверный рендеринг **`getServerSideProps`**

```
// pages/news.js
export async function getStaticProps() {
  const res = await fetch('https://api.example.com/news');
  const news = await res.json();

  return {
    props: { news },
  };
}

export default function NewsPage({ news }) {
  return (
    <ul>
      {news.map(item => (
        <li key={item.id}>{item.title}</li>
      ))}
    </ul>
  );
}
```

Что происходит если *НЕ меняя ничего в коде* просто заменить имя метода:

Имя метода **getStaticProps**	Имя метода **`getServerSideProps`**
Данные загружаются один раз при сборке проекта (npm run build). HTML страницы создаётся заранее и сохраняется. Пользователь получает быстро	Данные загружаются при каждом запросе пользователя. HTML страницы генерируется на сервере в реальном времени. Пользователь всегда получает

загружаемую страницу, *но данные могут устареть*.	актуальные данные, но загрузка может быть чуть медленнее.

getStaticPaths — генерация путей для динамических страниц

```
export async function getStaticPaths() {
  const res = await fetch('https://api.example.com/posts');
  const posts = await res.json();

  const paths = posts.map(post => ({
    params: { slug: post.slug },
  }));

  return { paths, fallback: false };
}
```

API Routes

Next.js позволяет создавать серверные функции прямо в `pages/api/`. Т.е. можно писать **серверный код прямо в проекте**, без необходимости поднимать отдельный backend. API Routes — это способ создавать серверные обработчики HTTP-запросов внутри Next.js.. Они живут в папке pages/api/ и работают как мини-сервер внутри твоего приложения.

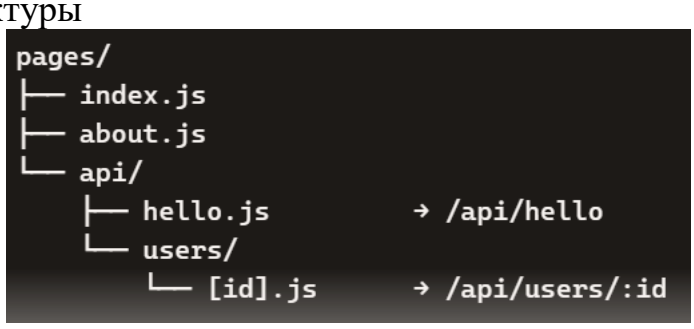
Каждый файл в pages/api/ становится отдельным эндпоинтом.

API Routes выполняются только на сервере, не попадают в клиентский JavaScript. Они не работают в static export (next export), потому что требуют сервер.

Что можно делать в API Routes

- Обрабатывать формы и отправку данных
- Работать с базами данных (например, через Prisma, MongoDB, PostgreSQL)
- Интегрироваться с внешними API (например, Telegram, Stripe, GitHub)
- Реализовать аутентификацию и авторизацию
- Загружать файлы
- Отправлять письма

Пример структуры



Простейший API Route

Когда пользователь делает запрос на /api/hello, вызывается функция handler. req — объект запроса (аналог Express.js). res — объект ответа.

```
// pages/api/hello.js
export default function handler(req, res) {
const { query, body } = req;

  console.log('Query:', query); // параметры URL
  console.log('Body:', body);   // тело запроса (например, JSON)
  res.status(200).json({ message: 'Привет из API!' });
}
```

Динамические API маршруты

```
// pages/api/users/[id].js
export default function handler(req, res) {
  const { id } = req.query;

  if (req.method === 'GET') {
    res.status(200).json({ userId: id });
  }
}
```

Запрос к /api/users/42 вернёт { userId: "42" }.

Обработка методов (GET, POST и др.)

```
// pages/api/user.js
export default function handler(req, res) {
  if (req.method === 'GET') {
    res.status(200).json({ name: 'Иван' });
  } else if (req.method === 'POST') {
    const { name } = req.body;
    res.status(201).json({ message: `Пользователь ${name} создан`
});
  } else {
    res.setHeader('Allow', ['GET', 'POST']);
    res.status(405).end(`Метод ${req.method} не разрешён`);
  }
}
```

Пример: API для формы обратной связи

```
// pages/api/contact.js
export default function handler(req, res) {
  if (req.method === 'POST') {
    const { name, email, message } = req.body;

    // Здесь можно отправить письмо, сохранить в БД и т.д.
    res.status(200).json({ success: true });
  } else {
    res.status(405).end();
  }
}
```

Оптимизация изображений

Компонент Image автоматически адаптирует размер, формат и качество изображения, автоматически улучшается производительность, качество изображений. Это гораздо “мощнее”, чем тег <img /> т.к.

- автоматически подбирает размер изображения под устройство пользователя,
- использует современные форматы (например, WebP),
- лениво загружает изображения (lazy loading),
- предотвращает визуальные скачки (layout shift),
- оптимизирует изображения на лету, даже если они хранятся на внешних серверах

```
import Image from 'next/image';

export default function Example() {
  return (
    <Image
      src="/images/photo.jpg"   // путь к изображению
      alt="Описание"
      width={800}               // ширина
      height={600}              // высота
      quality={80}              // качество (0-100)
      priority={true}           // приоритетная загрузка
      placeholder="blur"        // эффект размытия при загрузке
    />
  );
}
```

1.6. Создание серверной части приложений

Несмотря на то, что язык PHP был разработан Расмусом Лердорфом в 1994 году он является одним из самых простых и мощных скриптовых языков, предназначенных для разработки серверной части сайта. Изначально PHP представлял собой набор скриптов на другом языке Perl. Позже этот набор скриптов был переписан в интерпретатор на языке Си.

Считается, что название PHP произошло от словосочетания Personal Home Page Tools. Более современная аббревиатура PHP – “Препроцессор гипертекста”. Скрипты PHP обрабатывают на сервере запрос клиента и возвращают клиенту результат. Он является часто используемым языком для создания систем управления сайтом.

PHP представлял удобный набор инструментов для упрощенного создания веб-сайтов и веб-приложений. Какие преимущества предоставляет PHP?

Для всех наиболее распространенных операционных системам (Windows, MacOS, Linux) есть свои версии пакетов разработки на PHP, а это значит, что вы можете создавать веб-сайты на любой из этих операционных систем.

PHP может работать в связке с различными веб-серверами: Apache, Nginx, IIS

Простота и легкость освоения. Как правило, уже имея небольшой опыт в программировании на PHP, можно создавать простенькие веб-сайты

PHP похож на язык Си, поэтому, зная Си или один из языков с сиподобным синтаксисом, будет проще овладеть PHP

PHP поддерживает работу с множеством систем баз данных (MySQL, MSSQL, Oracle, Postgre, MongoDB и другие)

Распространенность хостинговых услуг и их дешевизна. Так как, как правило, хостинговые компании размещают веб-сайты на PHP на веб-серверах Apache или Nginx, которые работают на одной из операционных систем семейства Linux. И веб-серверы, и операционные системы на базе Linux бесплатны, что снижает общую стоимость использования хостинга

Постоянное развитие. PHP продолжает развиваться, выходят все новые версии, которые несут новые функции, адаптируя язык программирования к новым вызовам. И, как правило, перейти на новую версию не составляет труда.

1.6.1. Как PHP работает

Нужно соблюсти несколько требований, чтобы сервер мог выполнять программы на PHP. В конфигурационном файле WEB-Сервера необходимо добавить директиву обработки нужного типа файлов. Для функционирования PHP сам препроцессор должен быть установлен на сервере и верно сконфигурирован.

Иными словами. Что подразумевает установка PHP? Во-первых, нам нужен интерпретатор PHP. Во-вторых, необходим веб-сервер, например, Apache, с помощью которого мы сможем обращаться к ресурсам создаваемого нами сайта.

Код PHP может быть оформлен следующими способами:

```
<?php //стиль xml
//php инструкции
?>
```

В сокращенном варианте символы php после вопросительного знака отсутствуют, каждый скрипт открывается тегом <? и закрывается ?>. После этого скрипт может сохранятся и исполнятся в виде файла с расширением .php или помещаться внутрь html –документа. Интерпретатор с помощью тегов <?php ... ?> сможет понять, что весь текст между этими тегами следует рассматривать как код PHP. А весь код вне этих тегов рассматривается как код html.

Пример простого файла с PHP:

```
<html>
<head> </head>
<script language="php">
print "Hello,world-Первый способ вставки кода PHP<BR>";
</script>
<body>
Вставка кода PHP внутрь документа Html<BR>
<?
print "Hello, to you - второй способ вставки кода PHP<BR>";
?>
<?php
```

```
 echo "А это внутри PHP - третий способ вставки кода PHP<BR><HR>";
phpinfo(); //возвращает много информации о версии
// и настройках php
?>
</body>
</html>
```

Еще один пример html страницы, которая взаимодействует с серверной частью приложения на php:

```
<!DOCTYPE html>
<html>
<head>
<title>METANIT.COM</title>
<meta charset="utf-8">
</head>
<body>
<h2>Введи свои данные:</h2>
<form action="display.php" method="POST">
<p>Введите имя: <input type="text" name="firstname" /></p>
<p>Введите фамилию: <input type="text" name="lastname" /></p>
<input type="submit" value="Отправить">
</form>
</body>
</html>
```

Код html содержит форму с двумя текстовыми полями. При нажатии на кнопку данные этой формы отсылаются скрипту display.php, так как он указан в атрибуте action. Скрипт, который будет обрабатывать данные display.php.

```
<?php
$name = $_POST["firstname"];
$surname = $_POST["lastname"];
echo "Ваше имя: <b>".$name . " " . $surname . "</b>";
?>
```

$name - это переменная, которая будет хранить некоторое значение. Все переменные в PHP предваряются знаком $. И так как форма на странице index.html использует для отправки метод POST, то с помощью выражения $_POST["firstname"] мы можем получить значение, которое было введено в текстовое поле с атрибутом name="firstname". И это значение попадает в переменную $name.

С помощью оператора echo можно вывести на страницу любое значение или текст, которые идут после оператора. В данном случае (echo "Ваше имя: ".$name . " " . $surname . "") с помощью знака точки текст в кавычках соединяется со значениями переменных $name и $surname и выводится на страницу.

1.6.2. Переменные PHP

Переменные хранят отдельные значения, которые можно использовать в выражениях PHP. Для определения переменных применяется знак доллара $. Например $num; Здесь определена переменная $num. Поскольку определение переменной - это отдельная инструкция, она завершается точкой с запятой.

Как правило, названия переменный начинаются с маленькой буквы или символа подчеркивания. Стоит учитывать, что PHP является регистрозависимым языком, а значит, переменные $num и $Num будут представлять две разные переменные.

При наименовании переменных нам надо учитывать следующие правила:

Имена переменных должны начинаться с алфавитного символа или с подчеркивания

Имена переменных могут содержать только символы: a–z, A–Z, 0–9, и знак подчеркивания

Имена переменных не должны включать в себя пробелы

Для вывода значения переменной применялась команда echo, после которой указывалось выводимое значение. Однако есть и другой способ вывести значение переменной. Например, мы хотим одновременно вывести значения двух переменных:

```
<?php
$num_1 = 11;
$num_2 = 35;

echo "num_1 = $num_1  num_2=$num_2";
?>
```

Здесь функции echo передается строка. Чтобы встроить в строку значение переменной, в этой строке указываем имя переменной вместе со знаком $. И кода в строке PHP встретит выражение $num_1, он заменит это выражение значением переменной $num_1. То же самое касается и переменной $num_2. В итоге при выполнении этого скрипта браузер отобразит значения обеих переменных.

Пример создания таблицы умножения.

```
<table>
<?php
for ($i = 1; $i < 10; $i++)
{
    echo "<tr>";
    for ($j = 1; $j < 10; $j++)
    {
        echo "<td>" . $i * $j . "</td>";
    }
    echo "</tr>";
}
?>
</table>
```

Области видимости переменных PHP

Блоки циклов и условных конструкций не образуют отдельной области видимости, и переменные, определенные в этих блоках, мы можем использовать вне этих блоков:

Локальные переменные создаются внутри функции. К таким переменным можно обратиться только изнутри данной функции. То же самое относится и к параметрам функции: вне функции ее параметры также не существуют.

Статические переменные. Они отличаются тем, что после завершения работы функции их значение сохраняется. При каждом новом вызове функция использует ранее сохраненное значение. Чтобы указать, что переменная будет статической, к ней добавляется ключевое слово static.

Глобальные переменные по умолчанию не доступны внутри функции. Данный код не будет работать, а интрепретатор PHP известит нас, что переменная не определена.

Например:

```
<?php
$name = "Tom";
function hello()
{
    echo "Hello " . $name;
}
hello();
?>
```

Мы можем обратиться внутри функции к глобальной переменной. Для получения доступа к глобальной переменной в функции с помощью оператора global объявляется переменная с тем же именем. При чем мы можем не только получать ее значение, но и изменить его. В качестве альтернативы оператору global для обащения к глобальным переменным мы можем использовать встроенный массив $GLOBALS. Применяется выражение $GLOBALS["name"] - в квадратные скобки передается название переменной (без знака $).

```
<?php
$name = "Tom";
function changeName()
{
    global $name;
    $name = "Tomas";
}
changeName();
echo $name; // Tomas
?>
```

Проверка существования переменной PHP

Если переменная объявлена, но ей изначально не присвоено никакого значения (иначе говоря она не инициализирована), или если переменная вовсе неопределена, то нам будет проблематично ее использовать.

Ситуация может показатся искуственной. Тем не менее нередко переменные в PHP получают некоторые данные извне, например, ввод пользователя.

Соответственно возникает необходимость перед использованием данных проверить, что эти данные определены, что они имеются.

Для проверки существования переменной PHP предоставляет ряд встроенных функций.

isset() определяет, инициализирована ли переменная или нет (если переменная имеет значение null функция isset() также возвращает false).

empty() проверяет переменную на "пустоту". "Пустая" переменная - это переменная, значение которой равно null, 0, false или пустой строке - в этом случае функция empty() возвращает true:

1.6.3. Типы данных PHP

PHP является языком с динамической типизацией. Это значит, что тип данных переменной выводится во время выполнения, и в отличии от ряда других языков программирования в PHP не надо указывать перед переменной тип данных.

В PHP есть десять базовых типов данных:

- bool (логический тип)
- int (целые числа)
- float (дробные числа)
- string (строки)
- array (массивы)
- object (объекты)
- callable (функции)
- mixed (любой тип)
- resource (ресурсы)
- null (отсутствие значения)

Из этих типов данных первые четыре являются скалярными: bool, int, float, string.

Поскольку PHP - язык с динамической типизацией, то мы можем присваивать одной и той же переменной значения разных типов:

```
<?php
$id = 123;
echo "<p>id = $id</p>";
$id = "jhveruuyeru";
echo "<p>id = $id</p>";
?>
```

1.6.4. Типизация данных PHP

В ряде ситуаций – при определении свойств классов, параметров или возвращаемого значения функций PHP позволяет указать тип данных. Установка типа данных позволит избежать ситуаций, когда в программу будут переданы данные не тех типов, которые ожидалась разработчиком. Например, можно яв-

ным образом указать, что функция может принимать только число, то есть типизировать параметр функции.

```
function isPositive(int $number)
{
    return $number > 0;
}
$result1 = isPositive(25);                // норм - 25 число
$result2 = isPositive("25");               // норм - PHP может
преобразовать значение в число
$result3 = isPositive("-Youdontknowwhoiam"); // Ошибка TypeError
```

параметр $number должен представлять тип int, то есть целое число. Поэтому при вызове функции мы должны передать в функцию целочисленное значение. Если будет передано значение другого типа, то PHP попытается преобразовать значение. В некоторых случаях такое преобразование можно завершится успешно. других случаях преобразование может завершится неудачно, и программа завершит выполнение с ошибкой TypeError.

```
function sum(array $numbers, callable $condition)
```

Параметры этой функции должный представлять массив и другую функцию (тип callable). В качестве функции можно передать анонимную функцию.

В PHP 8 был добавлен тип union или объединение, который по сути представляет объединение типов, разделенных вертикальной чертой |. Например, мы хотим написать функцию сложения чисел, и чтобы в функцию можно было передавать только числа. Однако числа в PHP предствлены двумя типами - int и float. Чтобы не создавать по функции для каждого типа, применяют объединения.

```
function sum(int|float $n1, int|float $n2,): int|float
```

Ключевое слово static, добавленное в PHP 8, которое применяется, если надо возвратить из метода класса объект этого же класса:

Для установки типа возвращаемого из функции значения после списка параметров указывается двоеточие : и после него тип данных.

```
function isPositive (int $number) : bool{ return $number > 0;}
```

Типизация свойств объекта

В качестве типа свойств может применяться любой тип кроме callable. Стоит учитывать, что свойство, для которого не указан тип данных, по умолчанию имеет значение null. Тогда как свойство, для которого указан тип, неинициализировано, то есть не имеет никакого конкретного значения. Соответственно если нетипизированное свойство мы сможем использовать, то при попытке обратиться к типизированному, но неинициализиованному свойству программа завершит выполнение ошибкой:

```
class Person{

    public $name;       // равно null
    public int $age;    // неинициализировано
}
```

```
$tom = new Person();
echo $tom->name;      // норм - null
echo $tom->age;       // ошибка - свойство неинициализировано
$tom->name = "Tom";
$tom->age = 36;       // корректное значение
echo $tom->age;       // 36
```

1.6.5. Операции в PHP

Все арифметические операции – стандартны: % (получение остатка от деления), ** (возведение в степень).

Для объединения строк используется оператор "точка". Если переменные представляют не строки, а другие типы, например, числа, то их значения преобразуются в строки и затем также происходит операция объединения строк.

Оператор равенства и тождественности

Оба оператора сравнивают два выражения и возвращают true, если выражения равны. Но между ними есть различия. Если в операции равенства принимают два значения разных типов, то они приводятся к одному - тому, который интерпретатор найдет оптимальным. Например:

```
$a = (2 == "2");     // true (значения равны)
$b = (2 === "2");    // false (значения представляют разные типы)
```

Строка "2" по сути представляет то же значение, что и число 2, оператор сравнения возвратит true. Однако они представляют разные типы, поэтому оператор тождественности возвратит false

Аналогично работают операторы неравенства != и !==.

```
$a = (2 != "2");     // false, так как значения равны
$b = (2 !== "2");    // true, так как значения представляют разные типы
```

Отдельно стоит сказать про оператор <=>. Он также сравнивает два значения и возвращает

0, если оба значения равны

1, если значение слева больше, чем значение справа

–1, если значение слева меньше, чем значение справа

```
$a = 2 <=> 2;      // 0    (эквивалентно 2 == 2)
$b = 3 <=> 2;      // 1    (эквивалентно 3 > 2)
$c = 1 <=> 2;      // -1   (эквивалентно 1 < 2)
echo "a=$a   b=$b   c=$c";  // a=0  b=1  c=-1
```

1.6.6. Комбинированный режим HTML и PHP

Также мы можем использовать конструкцию if..else иным образом, переключаясь внутри конструкции на код HTML:

```
<!DOCTYPE html>
<html>
<head>
<title>METANIT.COM</title>
```

```
<meta charset="utf-8" />
</head>
<body>
<?php
$a = 5;
?>

<?php if ($a > 0) { ?>
<h2>Переменная a больше нуля</h2>
<?php } ?>

</body>
</html>
```

В данном случае само условие указывется в отдельном блоке php: <?php if ($a > 0) { ?>. Важно, что при этом этот блок содержит только открывающую фигурную скобку "{".

Завершается конструкция if другим блоком php, который содержит закрывающую фигурную скобку: <?php } ?>

Между этими двумя блоками php располагается код, который отображается на html-странице, если условие в if истинно. Причем этот код представляет именно код html, поэтому здесь можно разместить различные элементы html, как в данном случае элемент <h2>

При необходимости можно добавить выражения else и elseif:

```
<!DOCTYPE html>
<body>
<?php
$a = -5;
?>

<?php if ($a > 0) { ?>
<h2>Переменная a больше нуля</h2>
<?php } elseif($a < 0) { ?>
<h2>Переменная a меньше нуля</h2>
<?php } else { ?>
<h2>Переменная a равна нулю</h2>
<?php } ?>
</body>
</html>
```

Также можно применять альтернативный синтаксис:

```
<!DOCTYPE html>
<html>
<head>
<title>METANIT.COM</title>
<meta charset="utf-8" />
</head>
<body>
<?php
$a = 0;
```

```
?>

<?php if ($a > 0): ?>
<h2>Переменная a больше нуля</h2>
<?php elseif($a < 0): ?>
<h2>Переменная a меньше нуля</h2>
<?php else: ?>
<h2>Переменная a равна нулю</h2>
<?php endif; ?>
</body>
</html>
```

1.6.7. Оператор match PHP

Начиная с версии 8.0 в PHP была добавлена поддержка другой, похожей конструкции - match. Она позволяет оптимизировать конструкцию switch. Конструкция match также принимает некоторое выражение и сравнивает его с набором значений.

Например, пусть у нас есть следующий код:

```
$a = 2;
switch($a)
{
    case 1:
        $operation = "сложение";
        break;
    case 2:
        $operation = "вычитание";
        break;
    default:
        $operation = "действие по умолчанию";
        break;
}
echo $operation;
```

Перепишем этот пример с использованием match:

```
$a = 2;
$operation = match($a)
{
    1 => "сложение",
    2 => "вычитание",
    default => "действие по умолчанию",
};
echo $operation;
```

Итак, match в скобках также принимает некоторое сравниваемое выражение (в данном случае это переменная $a). Сам блок кода match также обертывается в фигурные скобки, однако в конце после закрывающей фигурной скобки необходимо поставить точку с запятой. А вместо операторов case просто указываются значения, с которыми сравнивается выражение.

Но в отличие от switch, конструкция match возвращает некоторый результат. Поэтому после каждого сравнимого значения ставится оператор =>, после которого идет возвращаемый результат.

То есть в данном случае переменная $a равна 2, поэтому в конструкции match будет выполнять блок

```
2 => "вычитание",
```

Этот блок установит в качестве возвращаемого результата строку "вычитание".

Поскольку конструкция match возвращает результат, то этот результат мы можем присвоить другой переменной:

```
$operation = match($a){
    //.................
}
```

В итоге в переменной $operation будет храниться строка "вычитание".

Также мы можем переписать предыдущий пример следующим образом:

```
$a = 2;
match($a)
{
    1 => $operation = "сложение",
    2 => $operation = "вычитание",
    default => $operation = "действие по умолчанию",
};
echo $operation;
```

Сравнение значений и типов в операторах switch и match

Стоит отметить важное отличие конструкции switch от match: switch сравнивает только значение, но не учитывает тип выражения. Тогда как match также учитывает тип сравниваемого выражения. Рассмотрим разницу на примере. Пусть есть следующая конструкция switch:

```
switch (8.0) {
  case "8.0":
    $result = "строка";
    break;
  case 8.0:
    $result = "число";
    break;
}
echo $result; // строка
```

В конструкцию switch передается в качестве выажения число 8.0, но с точки зрения внутренней логики конструкции switch это выражение также соответствует строке "8.0". Поэтому в данном случае будет выполняться блок

```
case "8.0":
    $result = "строка";
    break;
```

Теперь посмотрим, что будет в аналогичном примере с match:

```
match (8.0) {
  "8.0" => $result = "строка",
  8.0 => $result = "число"
};
```

```
echo $result; // число
```

Конструкция match также будет учитывать тип выражения, а тип в данном случае float, поэтому будет выполняться блок:

```
8.0 => $result = "число"
```

1.6.8. Массивы PHP

В качестве элементов массива могут выступать объекты любого типа.

Есть несколько способов определения массивов. Первый способ представляет использование функции array():

```
$numbers = array();
```

В данном случае определяется пустой массив $numbers.

Второй способ представляет использование квадратных скобок []:

```
$numbers = [];
```

При определении массива мы сразу можем передать ему начальные данные. Если применяются квадратные скобки, то элементы массива передаются внутри скобок:

```
$numbers = [1, 2, 3, 4];
```

Аналогичное определение массива с помощью функции array():

```
$numbers = array(1, 2, 3, 4);
```

Если мы хотим установить элемент по еще не существующему индексу, мы можем это сделать: $numbers[5] = 76; или $numbers[] = 25;

Для обращения к элементам массива применяются ключи. Ключ может представлять число или строку или одновременно и числа, и строки.

Перебор массива

Для перебора массива мы можем применять стандартный метод for:

```
for($i=0; $i < count($users); $i++)
```

Такой способ перебора не поможет, если индексы определяются вручную и отличаются они от соседних индексов не на единицу, а на произвольную величину. В этом случае мы можем использовать специальный цикл - foreach:

```
foreach($users as $element)
```

Цикл foreach позволяет извлекать не только значения, но и ключи элементов:

```
foreach($users as $key => $value)
```

1.6.9. Ассоциативные массивы PHP

Это подвид массивов, в которых, в отличие от обычных массивов, в качестве ключа применяются строки. При создании ассоциативного массива мы явным образом указываем ключ элемента, после которого идет оператор => и значение элемента. Например, создание ассоциативного массива с помощью функции array():

```
$words = array("red" => "красный", "blue" => "синий");
```

необязательно инициализировать переменную массива при ее определении. Можно, как с обычными массивами, добавлять элементы по ходу:

```
$ words ["green"] ="зеленый";
```

PHP позволяет использовать в одном массиве числовые и строковые индексы:

1.6.10. Оператор => PHP

Оператор => позволяет сопоставить ключ с определенным значением. Хотя при определении массива выше нам не требовался подобный оператор, тем не менее мы можем его применять. Например, следующий массив:

$numbers = [1, 4, 9, 16];

Будет аналогичен следующему массиву:

```
$numbers = [0=>1, 1=>4, 2=>9, 3=>16];
// $numbers = array(0=>1, 1=>4, 2=>9, 3=>16);
```

Каждый элемент определяется в следующем формате: ключ => значение

Этот оператор может понадобится, если хотим переопределить порядок индексов. Так, по умолчанию нумеация индексов начинается с нуля и каждый следующий элемент имеет индекс предыдущего элемента + 1. Оператор => же позволяет определить свои индексы вручную для элементов, необязательно с нуля и необязательно по порядку:

```
$numbers = [1=> 1, 2=> 4, 5=> 25, 4=> 16];
echo $numbers[2];   // 4
```

Также мы можем задать индекс лишь для одного элемента, тогда для последующих элементов индекс автоматически будет увеличиваться на единиицу:

```
$numbers = [4=> 16, 25, 36, 49, 64];
print_r($numbers);
```

В данном случае индексация начинается с числа 4 - это индекс элемента 16. Тогда для элемента 25 индексом будет число 5 и так далее.

Результат:

```
Array ( [4] => 16 [5] => 25 [6] => 36 [7] => 49 [8] => 64 )
```

1.6.11. функция print_r PHP

Чтобы получить полное наглядное представление о том, как в конкретном массиве сопоставляются ключи и значения элементов, можно использовать функцию print_r, в которую в качестве параметра передается массив:

```
<?php
$numbers = [1, 4, 9, 16];
$numbers[] = 25;
print_r($numbers);
?>
```

Вывод скрипта:

```
Array ( [0] => 1 [1] => 4 [2] => 9 [3] => 16 [4] => 25 )
```

Также следует отметить, что необязательно как-то специально инициализировать переменную массива - мы можем по ходу добавлять элементы в массив:

```
<?php
$numbers[] = 20;
$numbers[] = 120;
$numbers[] = 720;
print_r($numbers);  // Array ( [0] => 20 [1] => 120 [2] => 720 )
?>
```

1.6.12. Операции с массивами PHP

Рассмотрим некоторые встроенные наиболее распространенные функции, которые мы можем применять при работе с массивами.

is_array() проверяет, является ли переменная массивом

count() и sizeof() получают количество элементов массива:

shuffle перемешивает элементы массивы случайным образом

compact создает из набора переменных ассоциативный массив, где ключами будут имена переменных. Каждая переменная указывается в кавычка без знака $. Результатом функции является новый массив.

```
$model = "Apple II";
$producer = "Apple";
$year = 1978;
$data = compact("model", "producer", "year");
```

asort сортировка по возрастанию значений элементов

arsort сортировать массив в обратном порядке

ksort сортировка по ключам

natsort выполняет естественную сортировку

natcasesort сортировка без учёта регистра

С помощью дополнительного параметра можно явно указать интерпретатору PHP тип сортировки. Данный параметр может принимать три значения:

SORT_REGULAR: автоматический выбор сортировки

SORT_NUMERIC: числовая сортировка

SORT_STRING: сортировка по алфавиту

```
$states = ["Spain" => "Madrid", "France" => "Paris", "Germany" =>
"Berlin", ];
asort($states);
print_r($states);
// массив после asort   - сортировка по значениям элементов
// Array ( [Germany] => Berlin [Spain] => Madrid [France] => Paris )

ksort($states);
print_r($states);
// массив после ksort - сортировка по ключам элементов
//  Array ( [France] => Paris [Germany] => Berlin [Spain] => Madrid
)
```

Если значения представляют строки, то PHP сортирует по алфавиту. Однако подобная сортировка не учитывает числа и регистр. Поэтому значение "Windows 10" будет идти в самом начале, а не в конце, как должно было быть.

```
<?php
$os = array("Windows 7", "Windows 8", "Windows 10");
asort($os);
print_r($os);
// результат
// Array ( [2] => Windows 10 [0] => Windows 7 [1] => Windows 8 )
?>
```

Выполним естественную сортировку:

```
<?php
$os = array("Windows 7", "Windows 8", "Windows 10");
natsort($os);
print_r($os);
// результат
// Array ( [0] => Windows 7 [1] => Windows 8 [2] => Windows 10)
?>
```

Генераторы PHP

Генератор предоставляет функцию, которая генерирует набор значений.

Но естественно может возникнуть вопрос: а зачем нужны генераторы? Разве мы не можем с тем же успехом перебирать обычный массив?

Дело в том, что при работе с массивом весь массив загружается в память. При небольших объемах проблема может быть игнорироваться. Но чем больше размер массива, соответственно тем больше издержки и потери в производительности. Именно эту проблему и призваны решить генераторы, которые извлекают только одно значение одномоментно при обращении к функции, экономя тем самым вычислительные ресурсы.

```
$numbers = [1, 2, 3, 4, 5];
foreach($numbers as $number)
{
    echo $number; // 12345
}
```

Для возвращения значения из функции применяется оператор yield. Но в отличие от return оператор yield сохраняет состояние функции, позволяя ей продолжать работу с того места, когда остановилось ее выполнение. Можно перебирать результат функции генератора в цикле как стандартный массив. При переборе в цикле мы фактически перебираем результат функции как обычный массив, каждый элемент которого имеет числовой индекс, начиная с нуля. С помощью оператора from можно определять массив - источник данных для генератора.

```
function generateNumbers($start, $end)
{
    for($i = $start; $i < $end; $i++){
        yield $i;
    }
}
foreach(generateNumbers(4, 9) as $number)
```

```
{
    echo $number; // 45678
}
```

1.6.13. Функции в PHP

Общий синтаксис определения функции

```
function имя_функции([параметр [, ...]])
{
    // Инструкции
}
```

Можно дать определение функции при определенном условии:

```
if(true){
    function hello()
    {
        echo "Hello PHP";
    }

    hello();
}
```

Анонимные функции

Анонимные функции позволяют передавать в качестве параметров функции другие функции или присваивать их переменным. Анонимная функция определяется как обычная функция за тем исключением, что она не имеет имени. Например:

```
$hello = function($name)
{
    echo "<h2>Hello $name</h2>";
};
```

Здесь переменной $hello присваивается анонимная функция. Эта функция также определяется с помощью ключевого слова function. Она также принимает параметры - в данном случае параметр $name. И также она имеет некоторый блок операторов.

Для вызова подобной функции применяется имя представляющей ее переменной: $hello("Tom");

Распространенным случаем применения анонимных функций является передача их параметрам других функции – коллбеки (callback function).

```
<?php
function welcome($message)
{
    $message();
}
$goodMorning = function() { echo "<h3>Доброе утро</h3>"; };
$goodEvening = function() { echo "<h3>Добрый вечер</h3>"; };

welcome($goodMorning);          // Доброе утро
welcome($goodEvening);          // Добрый вечер
```

```
welcome(function(){ echo "<h3>Привет</h3>"; }); // Привет
?>
```

Стрелочные функции

Стрелочные функции (arrow function) позволяют упростить запись анонимных функций, которые возвращают некоторое значение. И при этом стрелочные функции автоматически имеют доступ к переменным из внешнего окружения. Стрелочная функция определяется с помощью оператора fn:

```
fn(параметры) => действия;
```

Стрелочные функции могут применяться в качестве параметров функции

```
function sum($numbers, $condition)
{
    $result = 0;
    foreach($numbers as $number){
        if($condition($number))
        {
            $result += $number;
        }
    }
    return $result;
}

$myNumbers = [-2, -1, 0, 1, 2, 3, 4, 5];

$positiveSum = sum($myNumbers, fn($n)=>$n > 0);
$evenSum = sum($myNumbers, fn($n) => $n % 2 === 0);
echo "Сумма положительных чисел: $positiveSum <br /> Сумма четных
чисел: $evenSum";
```

1.6.14. Замыкания в PHP

Замыкания в PHP представляют анонимную функцию, которая может использовать переменные из своего локального окружения. В отличие от обычных анонимных функций замыкания в PHP применяют выражение use.

По умолчанию внешние переменные для анонимной функции не существует. Из данной сиутации мы можем выйти, использовав оператор global или массив $GLOBALS, которые рассматриваются в последующих параграфах. Замыкания также позволяют решить эту проблему.

```
<?php
$number = 89;

$showNumber = function() use($number)
{
    echo $number;
};

$showNumber();
?>
```

Выражение use() получает внешние переменные, которые анонимная функция собирается использовать. И теперь при ее выполении браузер выведет значение переменной $number.

Подобным образом функция-замыкание может захватывать и большее количество внешних переменных, а также как и другие функции применять параметры:

```
$a = 8;
$b = 10;

$closure = function($c) use($a, $b)
{
    return $a + $b + $c;
};

$result = $closure(22); // 40
echo $result003B
```

1.6.15. Ссылки в PHP

Ссылки в PHP позволяют ссылаться на область памяти, где расположено значение переменной или параметра. Для создания ссылки перед переменной указывается символ амперсанда - &.

При передаче по ссылке перед параметром ставится знак амперсанда: function square(&$a). Теперь интерпретатор будет передавать не значение переменной, а ссылку на эту переменную в памяти. То есть теперь и переменная $number и параметр $a будут указывать на одну и ту же область в памяти. В итоге, значение переменной $number после передачи параметру &$a также будет изменено.

Функция также может возвращать ссылку. В этом случае при определении и вызове функции перед ее именем ставится знак амперсанда:

```
<?php
function &checkName(&$name)
{
    if($name === "admin") $name = "Tom";
    return $name;
}

$userName = "admin";
$checkedName = &checkName($userName);
echo "<br />userName: $userName";
echo "<br />checkedName: $checkedName";
?>
```

В данном случае функция checkName() получает параметр по ссылке и возвращает ссылку - фактически ссылку, которая передается в функции. Для имитации работы функция проверяет имя пользователя и изменяет его на некоторое стандартное, если оно равно "admin". После выполнения функции переменная $checkedName фактически будет содержать ссылку на переменную $userName.

1.6.16. Работа с данными в PHP, полученными от клиента

Получение данных из строки запроса. GET запросы

Самым простым способом передачи данных на сервер приложению PHP извне представляет передача данных через строку запроса.

Строка запроса представляет набор параметров, которые помещаются в адресе после вопросительного знака. При этом каждый параметр определяет название и значение. Друг от друга параметры отделяются знаком амперсанда. Например, в адресе:

```
http://localhost/user.php?name=Tom&age=36
```

Часть ?name=Tom&age=36 представляет строку запроса, в которой есть два параметра name и age. Для каждого параметра определено имя и значение, которые отделяются знаком равно. Параметр name имеет значение "Tom", а параметр age - значение 36.

В PHP по умолчанию определен глобальный ассоциативный массив $_GET, который хранит все значения, передаваемые в запроса GET. Используя ключи передаваемых данных, мы можем из массива $_GET получить передаваемые значения.

При отправки строки запроса ключами в этом массиве будут названия параметров, а значениями - значения параметров. Однако стоит учитывать, что в адресной строке необязательно будет использоваться строка запроса или конкретно данный параметр. Поэтому перед получением значения параметра сначала стоит проверить, а передан ли вообще такой параметр:

```
<?php
$name = "не определено";
$age = "не определен";
if(isset($_GET["name"])){

    $name = $_GET["name"];
}
if(isset($_GET["age"])){

    $age = $_GET["age"];
}
echo "Имя: $name <br> Возраст: $age";
?>
```

Получение данных из строки запроса. POST запросы

Одним из основных способов передачи данных является обработка форм которые содержат в себе различные элементы ввода - текстовые поля, кнопки и т.д. И с помощью заполнения формы мы можем ввести некоторые данные и отправить их на сервер. А сервер уже обрабатывает эти данные..

```
<form action="user.php" method="POST">
    <p>Имя: <input type="text" name="name" /></p>
    <p>Возраст: <input type="number" name="age" /></p>
    <input type="submit" value="Отправить">
</form>
```

Сайт может отправлять формы и запросом GET, в этом случае для получения тех же значений формы применяется массив $_GET.

Для обработки запросов типа POST в PHP используется встроенная глобальная переменная $_POST. Она представляет ассоциативный массив данных, переданных с помощью метода POST. Используя ключи, мы можем получить отправленные значения. Ключами в этом массиве являются значения атрибутов name у полей ввода формы.

Например, так как атрибут name поля ввода возраста имеет значение age (<input type="number" name="age" />), то в массиве $_POST значение этого поля будет представлять ключ "age": $_POST["age"]

Т.к. возможны ситуации, когда поле ввода будет не установлено, то в этом случае желательно перед обработкой данных проверять их наличие с помощью функции isset().

```
<?php
$name = "не определено";
$age = "не определен";
if(isset($_POST["name"])){

    $name = $_POST["name"];
}
if(isset($_POST["age"])){

    $age = $_POST["age"];
}
echo "Имя: $name <br> Возраст: $age";//ВНИМАНИЕ! НЕ БЕЗОПАСНО!!!
?>
```

Большое значение в серверной части приложения имеет организация безопасности данных. Рассмотрим несколько простых механизмов, которые могут повысить безопасность веб-сайта.

После отправки вредоносных данных в html разметку будет внедрен код javascript: echo $name

чтобы избежать подобных проблем с безопасностью, рекомендуется применять функцию htmlentities() или htmlspecialchars(). В качестве параметра она принимает значение, которое надо экранировать. Функция strip_tags() позволяет полностью исключить теги html.

```
$name = "не определено";
$age = "не определен";
if(isset($_POST["name"])){

    $name = htmlentities($_POST["name"]);
}
if(isset($_POST["age"])){

    $age = htmlentities($_POST["age"]);
}
echo "Имя: $name <br> Возраст: $age";
```

Отправка/получение массивов

Рассмотрим, как можно отправить на сервер и соответственно получить на сервере массивы данных.

Чтобы определить параметр строки запроса как массив, после названия параметра указываются квадраные скобки []. Затем мы можем присвоить некоторое значение: users[]=Tom. И сколько раз подобным образом будет присвоено значений, столько значений и будет в массиве. Все значения, как и обычно, отделяются амперсандом.

Название ключа передаваемых на сервер данных соответствует значению атрибута name у элемента формы. И чтобы указать, что какое-то поле ввода будет поставлять значение для массива, у атрибут name поля ввода в качестве значения принимает название массива с квадратными скобками. колько полей ввода с одним и тем же именем массива мы укажем, столько значений мы сможем передать на сервер. Так, в данном случае на сервер передается три значения в массиве users:

```
<form method="POST">
    <p>User 1: <input type="text" name="users[]" /></p>
    <p>User 2: <input type="text" name="users[]" /></p>
    <p>User 3: <input type="text" name="users[]" /></p>
    <input type="submit" value="Отправить">
</form>
```

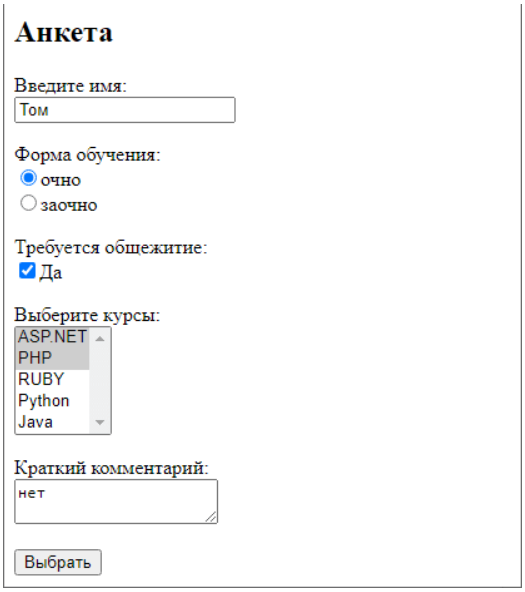
мы можем в элементах формы явным образом указать ключи:

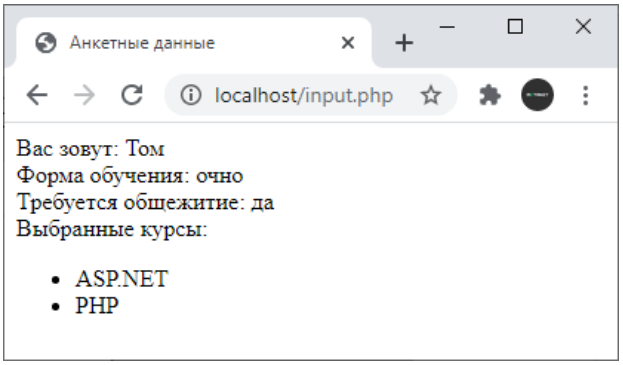
```
<form method="POST">
    <p>User 1: <input type="text" name="users[first]" /></p>
    <p>User 2: <input type="text" name="users[second]" /></p>
    <p>User 3: <input type="text" name="users[third]" /></p>
    <input type="submit" value="Отправить">
</form>

<?php
if(isset($_POST["users"])){

    $firstUser = $_POST["users"]["first"];
    $secondUser = $_POST["users"]["second"];
    $thirdUser = $_POST["users"]["third"];
    echo "$firstUser<br>$secondUser<br>$thirdUser";
}
?>
```

Рассмотрим комплексный пример обработки форм, в которой объединим обработку различных элементов html.






```
<!DOCTYPE html>
<html>
<head>
<title>Пример формы</title>
<meta charset="utf-8" />
</head>
<body>
<h2>Анкета</h2>
<form action="input.php" method="POST">
<p>Введите имя:<br>
<input type="text" name="firstname" /></p>
<p>Форма обучения: <br>
<input type="radio" name="eduform" value="очно" />очно <br>
<input type="radio" name="eduform" value="заочно" />заочно </p>
<p>Требуется общежитие:<br>
<input type="checkbox" name="hostel" />Да</p>
<p>Выберите курсы: <br>
<select name="courses[]" size="5" multiple="multiple">
    <option value="ASP.NET">ASP.NET</option>
    <option value="PHP">PHP</option>
    <option value="Ruby">RUBY</option>
    <option value="Python">Python</option>
    <option value="Java">Java</option>
</select></p>
<p>Краткий комментарий: <br>
<textarea name="comment" maxlength="200"></textarea></p>
<input type="submit" value="Отправить">
</form>
</body>
</html>
```

скрипт input.php, который будет обрабатывать эту форму

```
<?php
if(isset($_POST["firstname"]) && isset($_POST["eduform"]) &&
    isset($_POST["comment"]) && isset($_POST["courses"]))
```

```
{
    $name = htmlentities($_POST["firstname"]);
    $eduform = htmlentities($_POST["eduform"]);
    $hostel = "нет";
    if(isset($_POST["hostel"])) $hostel = "да";
    $comment = htmlentities($_POST["comment"]);
    $courses = $_POST["courses"];
    $output ="
    <html>
    <head>
    <title>Анкетные данные</title>
    </head>
    <body>
    Вас зовут: $name<br />
    Форма обучения: $eduform<br />
    Требуется общежитие: $hostel<br />
    Выбранные курсы:
    <ul>";
    foreach($courses as $item)
        $output.="<li>" . htmlentities($item) . "</li>";
    $output.="</ul></body></html>";
    echo $output;
}
else
{
    echo "Введенные данные некорректны";
}
?>
```

1.6.17. Отправка файлов на сервер и обработка в PHP

Чтобы загрузить файл на сервер, нам надо использовать форму с параметром enctype="multipart/form-data" и массив $_FILES.

Все загружаемые файлы попадают в ассоциативный массив $_FILES. Чтобы определить, а есть ли вообще загруженные файлы, можно использовать конструкцию if: if ($_FILES)

Массив $_FILES является двухмерным. Мы можем загрузить набор файлов, и каждый загруженный файл можно получить по ключу, который совпадает со значением атрибута name.

Так как элемент для загрузки файла на форме имеет name="filename", то данный файл мы можем получить с помощью выражения $_FILES["filename"].

У каждого объекта файла есть свои параметры, которые мы можем получить:

$_FILES["file"]["name"]: имя файла

$_FILES["file"]["type"]: тип содержимого файла, например, image/jpeg

$_FILES["file"]["size"]: размер файла в байтах

$_FILES["file"]["tmp_name"]: имя временного файла, сохраненного на сервере

$_FILES["file"]["error"]: код ошибки при загрузке

Также мы можем проверить наличие ошибок при загрузке. Если у нас нет ошибки, то поле $_FILES["filename"]["error"] содержит значение UPLOAD_ERR_OK.

При отправке файла на сервер он сначала загружается во временное место, из которого затем с помощью функции move_uploaded_file() он перемещается в каталог сервера, где расположен скрипт .php. Функция move_uploaded_file() принимает два параметра путь к загруженному временному файлу и путь, куда надо поместить загруженный файл. Также мы можем указать другой путь, например, допустим, на сервере есть папка "upload", тогда, чтобы загружать в нее файлы, необходимо указать соответствующий путь.

```
<!DOCTYPE html>
<html>
<head>
<title>METANIT.COM</title>
<meta charset="utf-8" />
</head>
<body>
<?php
if ($_FILES && $_FILES["filename"]["error"]== UPLOAD_ERR_OK)
{
    $name = $_FILES["filename"]["name"];
    move_uploaded_file($_FILES["filename"]["tmp_name"], $name);
    echo "Файл загружен";
}
?>
<h2>Загрузка файла</h2>
<form method="post" enctype="multipart/form-data">
Выберите файл: <input type="file" name="filename" size="10" /><br
/><br />
<input type="submit" value="Загрузить" />
</form>
</body>
</html>
```

Ограничения и настройка загрузки

Можно настроить параметры в файле конфигурации в файле php.ini

```
upload_max_filesize = 2M
```

Также мы можем настроить папку для временных загружаемых файлов.

```
upload_tmp_dir =
```

Изменим скрипт upload.php так, чтобы он поддерживал множественную загрузку. Каждое поле выбора файла имеет атрибут name="uploads[]", поэтому сервер будет рассматривать набор отправленных файлов как единый массив.

```
<!DOCTYPE html>
<html>
<head>
<title>Мультизагрузка</title>
<meta charset="utf-8" />
</head>
```

```
<body>
<?php
if($_FILES)
{
    foreach ($_FILES["uploads"]["error"] as $key => $error) {
        if ($error == UPLOAD_ERR_OK) {
            $tmp_name = $_FILES["uploads"]["tmp_name"][$key];
            $name = $_FILES["uploads"]["name"][$key];
            move_uploaded_file($tmp_name, "$name");
        }
    }
    echo "Файлы загружены";
}
?>
<h2>Загрузка файла</h2>
<form method="post" enctype="multipart/form-data">
    <input type="file" name="uploads[]" /><br />
    <input type="file" name="uploads[]" /><br />
    <input type="file" name="uploads[]" /><br />
    <input type="submit" value="Загрузить" />
</form>
</body>
</html>
```

2. ПРАКТИЧЕСКИЙ РАЗДЕЛ

2.1. Программа лабораторных занятий

Тема 2. Язык разметки гипертекста HTML (4 часа).

Форматирование текста, списков, таблиц. Создание и корректировка шаблонов страниц.

Манипуляции с объектами DOM. Перетаскивание объектов Drag and Drop.

Тема 3. Каскадные таблицы стилей (CSS) (4 часа).

Задание по адаптивной верстке страницы без использования внешних библиотек https://edummf.bsu.by/mod/assign/view.php?id=7268 .

Задание по верстке страницы с использованием Bootstrap https://edummf.bsu.by/mod/assign/view.php?id=2291

Тема 4. Язык JavaScript. (14 часов).

Создание динамических приложений-клиентов. Создание игр[1]: игра Быки-коровы с компьютером, игра "камень/ножницы/бумага" с компьютером, Игра крестики-нолики. Работа с двумерными массивами, игра "Сапёр". Работа с двумерными массивами

Работа с многомерными массивами – задание (Судоку helper)

Работа с методами массивов – таблица продаж фирмы.

Использование библиотеки React для создания клиентских приложений. Создание резюме в котором есть доступ ко всем выполненным заданиям в рамках изучения данной темы.

Тема 5. Создание серверной части приложений (10 часов).

Взаимодействие между страницами на клентской части приложения и серверным кодом на PHP.

2.2. Контролирущие задания и упражнения для лабораторных работ

Задача Table

Имеется сгенерированный функцией generateData список больших данных на несколько сотен тысяч строк. Предполагаем, что мы - владельцы фермы, у которой малый ассортимент товаров, но очень много покупателей.

В данных есть информация о продажах: кому, чего и сколько мы продали:

- `company` - какая компания у нас купила товар
- `product` - что купила
- `count` - количество единиц товара

[1] Образовательный портал механико-математического факультета БГУ [Электронный ресурс] https://edummf.bsu.by/course/view.php?id=32 .

Также нам доступен PRODUCT_PRICES с ценами за *одну единицу* каждого товара.

Отдел продаж компании хочет получить таблицу с данными о продажах и некоторые метрики по продуктам. Таблица может выглядеть так, если у нас будет только 2 доступных продукта:

Компания	Молоко (шт.)	Молоко (руб.)	Сыр (шт.)	Сыр (руб.)	Итого (руб.)
Рога и копыта	12	48	10	60	108
Евроопт	10	40	20	120	160
--	--	--	--	--	--
Итого	22	88	30	180	168
Среднее	11	44	15	90	134
Максимум	--	--	--	--	--
< back		[10] per page			forward >

Задание

1) Убрать "битые" данные. Данные валидны, если:

- Компания - не пустая строка
- Продукт - не пустая строка
- Количество - товара натуральное число

2) Построить таблицу.

В ней для каждой компании *единственная строка* с столбцами - названием всех возможных продуктов и ячейками - общим количеством и общей ценой на которую у нас купил этот товар эта компания.

Важно! подписи столбцов должны совпадать с названием товара в массиве. Поэтому их нельзя просто захардкодить. Надо прочитать как-то, эти ключи из данных. Будем полагать, что все возможные товары представлены объектом `PRODUCT_PRICES` (пока они называются `Молоко`, `Мороженое`, `Сыр` и `Масло` но это пока потом добавится Йогурт и еще что-то. При проверке работы добавятся еще несколько продуктов в `PRODUCT_PRICES`. Они также должны отобразиться в итоговой таблице.).

3)Посчитать и отобразить метрики и агрегации

Помимо сухих данных, нам [нашей компании] могут понадобится дополнительные метрики, например:

1. Какой максимум (минимум) единиц или итоговой цены товара мы продали компаниям
2. Среднее от всех проданных единиц (или общей стоимости) товара
3. Медиана от всех продаж товара
4. Сумма всех продаж
5. Какой процент от всех продаж составляет конкретная компания

Необходимо:

- Добавить в таблицу несколько столбцов для подсчета метрик по строкам.
- Добавить несколько строк в футер таблицы для подсчета агрегированных метрик по данным.

Какие метрики вы будете отображать, предлагается выбрать самостоятельно из представленных выше. Лучше всего задать выбранный список метрик в константах, чтобы его можно было легко поменять.

Рекомендация: Вам стоит написать в итоге 4 функции, каждая из которых по данным таблицы вычисляет некую метрику (агрегированное значение). В идеале эта ф-ция получает в качестве параметров таблицу и имя поля которому что-то там вычисляется сумма/среднее/количество/максимум/минимум

4) Добавить пагинацию. Когда строк много (вспоминайте крестики-нолики размером 5000x5000) браузеру может быть очень тяжело отобразить такое количество данных на одной странице.

Поэтому таблица должна быть разбита на несколько страниц (пагинация).

под таблицей или в футере таблицы есть :

1. Кнопка "Предыдущая страница"
2. Показатель сколько строк отображается в одной странице таблицы
    1. По умолчанию 10
    2. Можно выбрать 10, 50 и 100
3. Кнопка "Следующая страница"

5) Дополнительно (не обязательно)

Добавить возможность сортировки по столбцам (например, click на заголовке столбца приводит к сортировке) Убедитесь, что пользователь понимает, как именно сейчас сортированы данные.
Добавить возможность фильтрации по названию компании

Задача Filterable List

Используя `fetch` и `Promise`, отобразить список постов, которые можно скачать из API https://jsonplaceholder.typicode.com. Посты можно фильтровать по их заголовку.

Посты можно получить через API: `jsonplaceholder.typicode.com/posts` (ссылку можно просто вставить в браузер, потому что браузер по умолчанию делает GET запрос на вставленную ссылку)

Условия

- Приложение реактивно: список фильтруется "на лету". Пользователь не должен нажимать enter или на какие-нибудь кнопки, чтобы отфильтровать. Достаточно начать вводить заголовок и список сам изменяется.

- При фильтрации мы **должны сделать запрос к API** (т.е. запрещено использовать posts.filter): "дай мне только те посты, заголовок которых содержит введенную пользователем строку". Для этого используются `query параметры` (почитайте об этом), например, json-placeholder поддерживает `title_like`, чтобы фильтровать:
`https://jsonplaceholder.typicode.com/posts?title_like=sunt%20aut%20facere` (заметьте, как в параметрах закодированы пробелы, почитайте, как это можно сделать). Обязательно проверьте, что специальные символы уходят в запросе правильно, например `&`

- При вводе в инпут для фильтрации, должна быть задержка (`debounce` , почитайте, как это можно сделать), чтобы запрос не уходил на ввод *каждой буквы,* а только когда пользователь остановится на значительное время (500ms)

- Использовать async/await хотя бы однажды

- Использовать цепочку then/catch хотя бы однажды

- Список выглядит приятно на экране

- Необходимо правильно разбить приложение на "слои" - файлы. **Рендеринг – отдельно, слой данных – отдельно.** Обязателен файл main.js. Необходимо использовать es modules.

Задавайте себе вопрос: чья ответственность делать те или иные вещи: пример, кто знает про DOM элементы? Ответ просится сразу: ну это точно не api модуль, наверное тот, кто работает с dom, наверное renderer. Кто должен знать, как и когда скачать данные? Точно не рендерер, потому что он рендерит. Что когда происходит – не его дело. Api только скачивает данные. Когда – не его дело. Получается, что main – это то место, где должно происходить скачивание и хранение данных.

Дополнительно

- Реализовать виртуализацию (infinite scroll) по 5 10 50 постов на одной "странице" (размер страницы предлагается определить константой или select'ом). Когда пользователь скроллит вниз, подгружаются следующие N постов. (Некоторые варианты решения: https://habr.com/ru/sandbox/189534/)

Для того, чтобы данные скачивать не все с самого первого, а "постранично", читаем https://github.com/typicode/json-server?tab=readme-ov-file#paginate

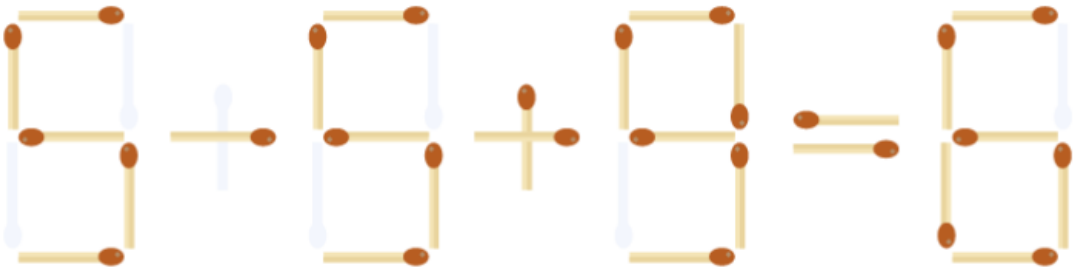
- Отобразить карточку пользователя (его имя, телефон и email) вместе с каждым постом этого пользователя

- Отдельного пользователя можно скачать по роуту `https://jsonplaceholder.typicode.com/users/USER_ID`, где вместо USER_ID нужно вставить id конкретного пользователя

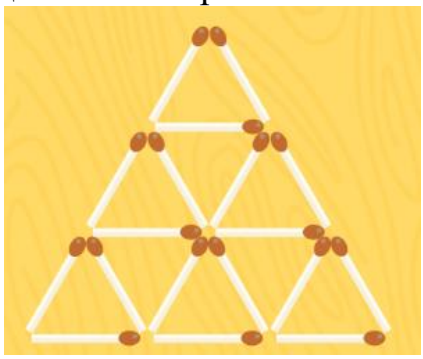
Перетаскивание изображений. Создание обучающей игры

Создать html-страницы с играми. При реализации использовать перемещение изображений.

1. Головоломка из спичек. Такие головоломки используются при обучении младших школьников. По условию задачи даётся неверное равенство 5 – 5 + 9 = 6, сложенное из спичек. Требуется переложить одну спичку таким образом, чтобы получилось верное равенство. Места, куда можно переместить спички обозначать изображением контура спички, например, как на рисунке ниже.

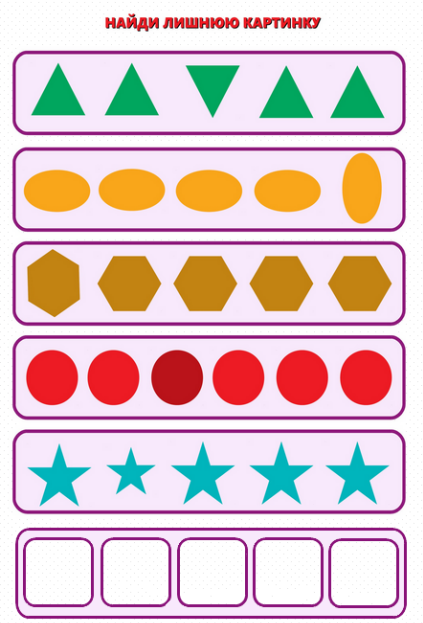


2. Задача со спичками. Уберите пять спичек так, чтобы осталось только 5 равных треугольников. Пользователь должен иметь возможность переместить убираемую спичку, например, в корзину. При необходимости пользователь может взять спичку и перетащить ее из корзины на место.



3. Перетяни карточки. Разместите на странице изображения предметов, в которых есть однотипные части, которые можно посчитать. Например, аквариумы с рыбками, клетки с мышками, корзинки с фруктами, или как на картинке ниже, пиццы с определенным числом одинаковых элементов начинки. Суть игры состоит в том, что пользователь должен переместить в блок под изображением предмета карточку с соответствующей цифрой. Должна быть возможность вернуть карточку с цифрой на первоначальное место и перетянуть под изображение предмета карточку с другой цифрой.

4. Найди лишнюю картинку. Условие задачи: разместить на странице пять рядов изображений. В каждом ряду разместить пять изображений: четыре одинаковых и одно очень похожее на них, но с небольшими отличиями. Пользователь должен иметь возможность убрать из ряда любое изображение. Примерный интерфейс приводится на картинке ниже.



Задания по React

TODO list

Материалы к заданию

Как отображать списки в React https://react.dev/learn#rendering-lists

Как правильно разбивать приложение на компоненты https://react.dev/learn/thinking-in-react

Состояние и жизненный цикл классовых компонентов https://ru.legacy.reactjs.org/docs/state-and-lifecycle.html

Пропсы классовых компонентов https://ru.legacy.reactjs.org/docs/components-and-props.html

Задача

Реализовать TODO list согласно следующим условиям:

1. Сверху страницы есть счетчики:
1. итоговое количество задач;
2. количество выполненных задач;
3. количество невыполненных задач.
2. Есть поле ввода и кнопка "Добавить", позволяющие создать новую задачу;
3. Задача имеет чекбокс – статус задачи: выполнена или нет;
4. Задачу можно удалить;
5. Выполненная задача отображается после невыполненной.

Требования

1. Отображение задачи должно быть вынесено в отдельный компонент;
2. Имя задачи не может: Быть пустым; Начинаться с пробельных символов; Заканчиваться пробельными символами.
4. Возможность удалить задачу появляется при наведении указателем мыши на задачу;
5. Код приложения выдержан в едином стиле;
6. Названия классов, функций и переменных лаконичны и отражают их

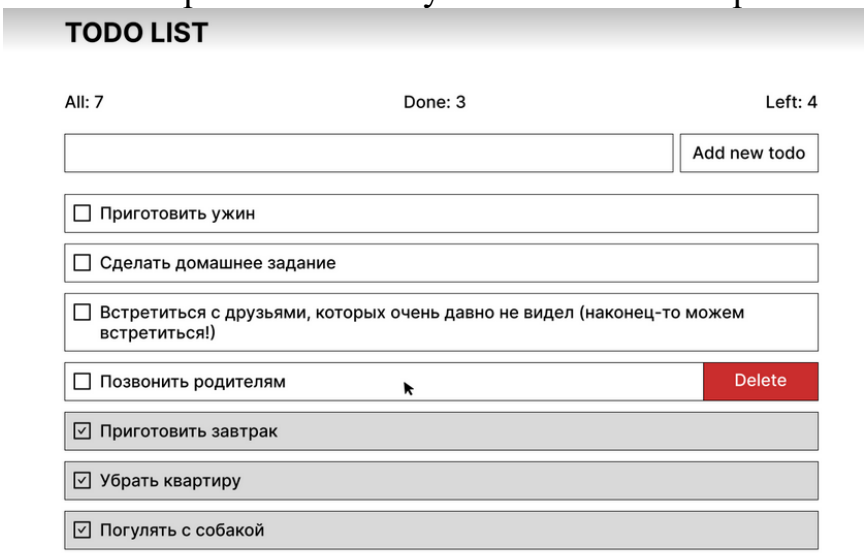
суть;

7. Код приложения не содержит лишних комментариев, файлов и лишнего кода, не используемого в приложении;

8. Приложение выглядит аккуратно.

Макет

Примерный макет приложения. Улучшения UI и UX приветствуются.



Дополнительно

Необходимо написать комментарий в главной компоненте приложения о том, какие дополнительные задачи были выполнены.

Приложение адаптивно под различные размеры экрана

На мобильном устройстве нет такого понятия, как "hover" (наведение указателем мыши), поэтому для удаления задачи необходимо придумать другой подход, или унифицировать оба подхода.

Название задачи можно редактировать;

Порядок задач можно редактировать;

Задание: Filterable TODO List

Материалы

Основные хуки:

useState https://react.dev/reference/react/useState

useMemo https://react.dev/reference/react/useMemo

useCallback https://react.dev/reference/react/useCallback

useEffect https://react.dev/reference/react/useEffect

Задача

Реализовать Filterable TODO List согласно следующим условиям:

1. Есть поле ввода и кнопка "Добавить", позволяющие создать новую задачу;

2. Задача имеет чекбокс – статус задачи: выполнена или нет;

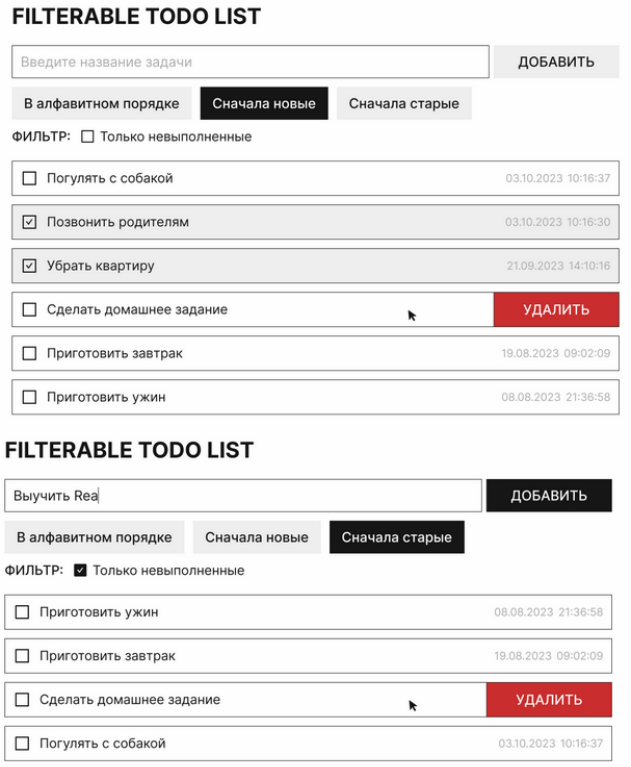
3. Задача имеет дату создания и отображает её;

4. Задачу можно удалить;

Возможность удалить задачу появляется при наведении указателем мыши на задачу;

5. Список имеет фильтр "Только невыполненные":
1. Представлен чекбоксом;
2. Включен – список отображает только невыполненные задачи;
3. Выключен – список отображает все задачи;
4. Выключен по умолчанию.
6. Список имеет сортировки:
1. "Сначала новые" – в порядке возрастания даты создания;
2. "Сначала старые" – в порядке убывания даты создания;
3. "В алфавитном порядке".
7. Только одна сортировка может быть активной одновременно;
8. Сортировка "Сначала новые" активна по умолчанию;
9. Активная сортировка визуально выделена.

Требования
1. НЕ разрешается использовать классовые компоненты;
2. Приложение грамотно разбито на компоненты и файлы;
3. Код приложения выдержан в едином стиле (см. General > Styleguide);
4. Код приложения не содержит лишних комментариев, файлов и неиспользуемого кода;
5. Названия классов, функций и переменных лаконичны и отражают их суть (см. General > Styleguide);
6. Интерфейс (UI) приложения выглядит аккуратно.
Макет
Примерный макет приложения. Улучшения UI и UX приветствуются.



Дополнительно
Приложение адаптивно под различные размеры экрана;
Фильтр "Только невыполненные" представлен тумблером (toggle ,switch);

Сортировки отображены SVG иконками;
Задачи не очищаются после перезагрузки страницы (см.localStorage).

3. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ

3.1. Перечень рекомендуемых средств диагностики

Диагностика результатов учебной деятельности по дисциплине «Введение в веб-программирование» проводится, как правило, во время аудиторных занятий. Для диагностики используются:

– устный опрос;

– отчет по лабораторным и домашним заданиям.

Оценка за ответы на лекциях (опрос) и лабораторных занятиях включает в себя полноту ответа, наличие аргументов, примеров из прак-тики и т. д.

Оценка отчета по лабораторным и домашним заданиям включает актуальность исследуемой проблемы, корректность используемых ме-тодов исследования, привлечение знаний из сопредельных областей, организация работы группы.

Контроль УСР проводится преподавателем с использованием ИКТ в форме опроса и проверки результатов выполнения работы.

Формой текущей аттестации по дисциплине «Введение в веб-программирование» учебным планом предусмотрен зачет.

3.2. Примерный перечень заданий для управляемой самостоятельной работы студентов

Тема 4. Язык JavaScript. (4 часа для специальности 6-05-0533-07 Математика и компьютерные науки, 2 часа для специальности 6-05-0533-08 Компьютерная математика и системный анализ)

Студентам предлагается выполнить одно из следующих заданий:

1. Создание Web-приложения по теме. Студентам необходимо разработать проект сайта и создать SPA сайт или многостраничный сайт, используя JavaScript, запросы на сервер, современные подходы к верстке. Предлагаемые для разработки темы приложений:
   - компьютерный клавиатурный тренажер
   - новостной сайт с подключаемыми с других сайтов информерами (курсы валют, прогноз погоды, спортивные новости, анекдоты и т.п.).
   - Интернет-галерея фотографий с возможностью оценивания. Очередность отображения фотографий зависит от рейтинга.
   - Интернет-магазин (по продаже цветов, по продаже компьютерной техники, по продаже настольных игр и др.).
2. Создание простой интерактивной игры (судоку, 15, угадай число, морской бой, крестики-нолики, сапер).
3. Создание страницы, визуализирующей построение треугольника по трем элементам которые интерактивно задаются (3 стороны, 2 стороны и угол, 2 угла и сторона, высота и 2 стороны и т.п.).

Форма контроля - защита практических заданий

Тема 3. Использование библиотек для создания клиентских приложений (4ч).

Студентам предлагается выполнить одно из следующих заданий:

1. Реализовать свой блог, используя Next.js. Блог – это набор предсгенерированных HTML страниц. Блог доступен онлайн, для этого можно использовать, например, https://vercel.com/ или https://www.netlify.com/; HTML страницы проще всего генерировать из markdown файлов. Библиотеку для парсинга и конвертации .md в HTML предлагается выбрать самостоятельно.
2. Реализовать TODO List используя SolidJS https://www.solidjs.com/ Требования: Приложение позволяет создать, удалить и редактировать todo; Todo не сбрасываются после перезагрузки страницы.
3. Выполнить задание по использованию Redux в TodoList или Notion, используя Redux-Toolkit

Форма контроля - защита практических заданий.

Тема 5. Создание серверной части приложений. (2 часа)

Студентам предлагается выполнить одно из следующих заданий:

1. Поздравления. По заданному в файле списку фамилий напечатать каждому упомянутому в списке поздравление к определенному празднику. Перечень желаемых “благ” выбирать как случайное подмножество из заготовленного списка (например, здоровья, счастья, продвижения по службе, долголетия и т.д.) который хранится в БД.
2. Приглашения. Имеется список членов коллектива с указанием принадлежности каждого к различным общественным организациям (профком, ученый совет, общество книголюбов, секция футбола и т. д.). Напечатать приглашение всем членам на очередное заседание указанной организации. Задается только вид организации, место и время сбора.
3. Игра. Создать простую игру типа: “Угадай задуманное число”,”камень-ножницы-бумага”, “black jack” и т.п.
4. Спроектировать и создать простую базу данных из 2-3 таблиц. Например, «Видеотека», «Сувениры», «Заказы», «Достопримечательности». Организовать средствами PHP соединение с БД, выполнить запрос на извлечение информации из БД, когда клиент в форме на html странице выбирает желаемые параметры. Результаты выполнения запроса передать клиенту и отобразить. Создать класс на добавление информации в БД.

Форма контроля - защита практических заданий.

3.3. Описание инновационных подходов и методов к преподаванию учебной дисциплины

При организации образовательного процесса используются

• ***методы и приемы развития критического мышления***, которые представляют собой систему, формирующую навыки работы с информацией в процессе чтения и письма; понимании информации как отправного, а не конечного пункта критического мышления;

• ***практико-ориентированный подход***, который предполагает освоение содержание образования через решения практических задач.

3.4. Методические рекомендации по организации самостоятельной работы обучающихся

При изучении учебной дисциплины рекомендуется использовать следующие формы самостоятельной работы:

• изучение литературы и материалов электронных источников по проблемам дисциплины;

• выполнение домашнего задания;

• подготовка к лабораторным занятиям;

• курсовые, дипломные и научно-исследовательские работы, связанные с тематикой дисциплины;

• подготовка к участию в конференциях с докладами по проблемам дисциплины.

Для организации самостоятельной работы студентов по учебной дисциплине рекомендуется использовать современные информационные ресурсы, размещенные на образовательном портале смешанного и дистанционного обучения БГУ, содержащие учебные материалы (курс лекций, задания к лабораторным и домашним работам и т. д.)

3.5. Примерный перечень вопросов к зачету

1. Протоколы интернет tcp/ip, http, https, ftp. Характеристика, назначение.
2. Маршрутизация, хостинг.
3. Html-документ и его основное назначение. Структура html-документа. Основные теги использующиеся в html-документе.
4. Сформулируйте основные отличия html5 от html.
5. Общие подходы к дизайну сайта. Разработка макета страницы.
6. Каково назначение CSS? Структура CSS файла. Как связать файл стилей и конкретный html-документ?
7. Различие между стилем в CSS и свойством в html? Есть ли отличия в использовании стилей и свойств?
8. Для каких целей используется набор правил в CSS? Что такое класс в CSS и как его правильно использовать?
9. Назначение селектора при формировании таблицы стилей.

10. Какие способы форматирования текста можно применить в таблице для более компактного размещения информации?
11. Как формируются блоки в html-документе?
12. В чем различие между полями и отступами?
13. Как можно спрятать блок на веб-странице?
14. Как выровнять блок по вертикали и горизонтали в контейнере?
15. Использование бибилиотек стилей. Преимущества и недостатки.
16. Методы адаптивной верстки веб-страницы.
17. Преимущества и ограничения программ, работающих на стороне клиента.
18. Различие между статическим и динамическим контентом на веб-странице. Возможности и ограничения JavaScript в браузере. Размещение JavaScript кода на веб-странице. Модули.
19. Объектная модель html страницы. Работа с DOM в JavaScript.
20. Различия var/let/const при объявлении идентификаторов.
21. Типы данных в JavaScript. JavaScript. Преобразования и приведение типов. Иммутабельность примитивных данных.
22. Преобразование типов. Использование знака плюс в JavaScript.
23. Массивы в JavaScript. Методы массивов.
24. Строки. Методы для работы со строками.
25. "Обычные"/"стрелочные" функции в JavaScript.
26. Контекст (this) в JavaScript и его связь с функциями.
27. События в JavaScript. Способы назначения обработчиков событий. Фазы всплытие/погружение событий. Делегирование событий.
28. Классы в JavaScript. Конструктор, поле, метод, статический метод.
29. Асинхронность в JS: концепция Event Loop.
30. Использование асинхронности в JS: Promise. Методы then, catch, finally. Async-await. Fetch.
31. Серверная технология PHP. Скрипты php. Основные отличия синтаксиса php от js.
32. Передача данных из клиентской части приложения. Обработка данных форм.

4. ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ

4.1. Список рекомендуемой литературы

Основная

1. Барвенов, С. А. Веб-программирование : электронный учебно-методический комплекс для специальности: 6-05-0533-07 «Математика и компьютерные науки» / Барвенов С. А., Романчик В. С. ; БГУ, Механико-математический фак., Каф. веб-технологий и компьютерного моделирования. - Минск : БГУ, 2024. - 1 электрон. опт. диск (CD-ROM). – URL: https://elib.bsu.by/handle/123456789/321284
2. Диков, А. В. Клиентские технологии веб-дизайна. HTML5 и CSS3 : учебное пособие для вузов / А. В. Диков. - Санкт-Петербург: Лань, 2023. - 184 с. - URL: https://lanbook.com/catalog/informatika/klientskie-tekhnologii-veb-dizayna-html5-i-css/.
3. Дронов В. А. JavaScript: 20 уроков для начинающих / В.А. Дронов. - Санкт-Петербург : БХВ-Петербург, 2022. - 352 с. - URL: https://ibooks.ru/bookshelf/385776.
4. Заяц, А. М. Заяц, А. М. Проектирование и разработка Web-приложений : введение в Frontend и Backend разработку на JavaScript и Node.Js : учебное пособие / А. М. Заяц, Н. П. Васильев. - Санкт-Петербург ; Москва ; Краснодар : Лань, 2024. - 118 с. – URL: https://e.lanbook.com/book/154380
5. Никсон, Р. Создаем динамические веб-сайты с помощью PHP, MySQL, JavaScript, CSS и HTML 5 / Робин Никсон ; [пер. с англ. С. Черников]. - 6-е изд. ; включает React, PHP 8 & MySQL 8 . - Санкт-Петербург ; Москва ; Минск : Питер, 2023. - 830 с. - URL: https://ibooks.ru/bookshelf/386792/reading.
6. Дронов, В. А. React 17. Разработка веб-приложений на JavaScript / Владимир Дронов. - Санкт-Петербург : БХВ-Петербург, 2022. - 384 с. - URL: https://ibooks.ru/bookshelf/385763/reading.

Дополнительная

1. Хрусталев А. HTML5 + CSS3. Основы современного WEB-дизайна / А. Хрусталев, А. Кириченко.– СПб: Символ-Плюс, 2018. – 346 с.
2. Грант К. CSS для профи/ Кит Грант. – СПб: Питер, 2021. - 496 с.
3. Фрэйн Б. HTML5 и CSS3. Разработка сайтов для любых браузеров и устройств Пер. с англ., СПб: Питер, 2017. – 272 с.
4. Диков, А. В. Web-программирование на JavaScript: учебное пособие / А. В. Диков. - Санкт-Петербург: Лань, 2025. - 168 с. - URL: h https://lanbook.com/catalog/informatika/web-programmirovanie-na-javascript/.
5. Диков, А. В. Клиентские технологии веб-программирования: JavaScript и DOM : учебное пособие / А. В. Диков. - Санкт-Петербург ; Москва ; Краснодар : Лань, 2020. - 121 с. - URL: https://reader.lanbook.com/book/126934.

6. Робсон, Э. Изучаем HTML, XHTML и CSS / Элизабет Робсон, Эрик Фримен ; [пер. с англ. В. Черник]. - 2-е изд. - Москва ; Санкт-Петербург ; Минск : Питер, 2022. - 720 с. - URL: https://ibooks.ru/bookshelf/377028/reading.

7. Янцев, В. В. JavaScript. Картинки, галереи, слайдеры : учебное пособие / В. В. Янцев. - Санкт-Петербург ; Москва ; Краснодар : Лань, 2022. - 250 с. - URL: https://reader.lanbook.com/book/256067.

8. Колисниченко, Д. Разработка веб-приложений на PHP 8/ Д. Колисниченко – BHV-СПб.: 2024. – 496с.

9. Татро, К. Создаем динамические веб-сайты на PHP = Programming PHP. Creating Dynamic Web Pages / Кевин Татро, Питер Макинтайр ; [пер. с англ. Е. Матвеева]. - 4-е междунар. изд. - Санкт-Петербург ; Москва ; Минск : Питер, 2021. - 541 с. - URL: https://ibooks.ru/bookshelf/376967/reading.

4.2. Электронные ресурсы

1. Образовательный портал механико-математического факультета. Веб-программирование КМ+Произв. [Электронный ресурс]. – Режим доступа: https://edummf.bsu.by/course/view.php?id=32 – Дата доступа: 29.01.2025.

2. Образовательный портал механико-математического факультета. Веб-программирование 1 курс, 2,6,9 группа (2 семестр [Электронный ресурс]. – Режим доступа: https://edummf.bsu.by/course/view.php?id=619 – Дата доступа: 29.01.2025.

3. Образовательный портал механико-математического факультета. Web - программирование. II курс (React) [Электронный ресурс]. – Режим доступа: https://edummf.bsu.by/course/view.php?id=1100 – Дата доступа: 29.01.2025.

4.3. Учебно-методическая карта учебной дисциплины
Очная форма получения высшего образования

для специальности 6-05-0533-07 Математика и компьютерные науки

Номер раздела, темы	Название раздела, темы	Количество аудиторных часов					Количество часов УСР	Формы контроля знаний
		Лекции	Практические занятия	Семинарские занятия	Лабораторные занятия	Иное		
1	Введение	2						Опрос
2.	Язык разметки гипертекста HTML	2			4			Опрос, защита практических заданий, электронные контрольные тесты
3.	Каскадные таблицы стилей (CSS)	2			4			Опрос, отчет по лабораторным работам, защита практических заданий, электронные контрольные тесты
4.	Язык JavaScript	20			16		4	Опрос, защита практических за-даний, отчет по лабораторным работам, электронные контрольные тесты
	Создание серверной части приложений	10			8		2	Опрос, защита практических за-даний,
	ВСЕГО: 108 часов	36			30		6	Зачет

для специальности 6-05-0533-08
Компьютерная математика и системный анализ

Номер раздела, темы	Название раздела, темы	Количество аудиторных часов					Количество часов УСР	Формы контроля знаний
		Лекции	Практические занятия	Семинарские занятия	Лабораторные занятия	Иное		
1	Введение	2						Опрос
2.	Язык разметки гипертекста HTML	2			4			Опрос, защита практических заданий, электронные контрольные тесты
3.	Каскадные таблицы стилей (CSS)	2			4			Опрос, отчет по лабораторным работам, защита практических заданий, электронные контрольные тесты
4.	Язык JavaScript	20			16		2	Опрос, защита практических за-даний, отчет по лабораторным работам, электронные контрольные тесты
	Создание серверной части приложений	10			8		2	Опрос, защита практических за-даний,
	ВСЕГО: 110 часов	36			32		4	Зачет