*Содержание*

[**Методология БЭМ** 5](#_Toc119264619)

[**Группировка файлов** 5](#_Toc119264620)

[**Структура для блоков** 5](#_Toc119264621)

[**Структура для элементов** 6](#_Toc119264622)

[**Структура для модификаторов** 6](#_Toc119264623)

[**Теория** 7](#_Toc119264624)

[**HTML & CSS** 8](#_Toc119264625)

[**Описание сайта для браузеров и поисковых машин** 9](#_Toc119264626)

[**Метатеги для поисковиков** 9](#_Toc119264627)

[**Подключение шрифтов к странице** 9](#_Toc119264628)

[**Разнообразие контейнеров HTML5** 10](#_Toc119264629)

[**Списки** 10](#_Toc119264630)

[**Тег статьи** 10](#_Toc119264631)

[**Цитаты** 10](#_Toc119264632)

[**Тег <table>** 10](#_Toc119264633)

[**Тег <form>** 11](#_Toc119264634)

[**Поля ввода с тегом input** 11](#_Toc119264635)

[**Минимальные и максимальные значения** 11](#_Toc119264636)

[**Поля загрузки, сброса и отправки данных** 12](#_Toc119264637)

[**Поля ввода с другим синтаксисом** 12](#_Toc119264638)

[**Ярлыки** 12](#_Toc119264639)

[**Передаваемые значения** 12](#_Toc119264640)

[**Поля множественного и единичного выбора radio checkbox** 13](#_Toc119264641)

[**Стилизация переключателей и полей выбора** 13](#_Toc119264642)

[**Подсказки в текстовых полях placeholder** 14](#_Toc119264643)

[**Стилизация placeholder** 14](#_Toc119264644)

[**Обязательные поля required** 14](#_Toc119264645)

[**Особенности стилизации текстовых полей ввода, кнопок и ярлыков** 14](#_Toc119264646)

[**Обработчик поля input** 15](#_Toc119264647)

[**iframe** 15](#_Toc119264648)

[**Тег <video>** 15](#_Toc119264649)

[**Атрибуты тега <video>** 16](#_Toc119264650)

[**Тег <audio>** 16](#_Toc119264651)

[**Селекторы** 16](#_Toc119264652)

[**Селекторы атрибутов** 16](#_Toc119264653)

[**Селектор первого соседа** 17](#_Toc119264654)

[**Псевдоклассы** 17](#_Toc119264655)

[**Функциональные псевдоклассы** 17](#_Toc119264656)

[**Псевдоэлементы и селекторы атрибутов** 17](#_Toc119264657)

[**Наследственность в CSS** 18](#_Toc119264658)

[**Позиционирование элементов.** 18](#_Toc119264659)

[**Относительное позиционирование** 18](#_Toc119264660)

[**Фиксированное позиционирование** 18](#_Toc119264661)

[**Абсолютное позиционирование** 18](#_Toc119264662)

[**z-index** 19](#_Toc119264663)

[**Разделение файлов по типам** 20](#_Toc119264664)

[**CSS-директивы. @import** 20](#_Toc119264665)

[**Внешний и внутренний фокус элемента** 20](#_Toc119264666)

[**Блочные + строчные** 20](#_Toc119264667)

[**Свойства фона** 21](#_Toc119264668)

[**Расположение элементов по центру: margin: auto;** 21](#_Toc119264669)

[**Центрировать по вертикали** 21](#_Toc119264670)

[**Тени** 21](#_Toc119264671)

[**Линейный градиент** 22](#_Toc119264672)

[**Радиальный градиент** 23](#_Toc119264673)

[**Переполнение блоков** 24](#_Toc119264674)

[**Трансформации** 24](#_Toc119264675)

[**Плавность переходов** 25](#_Toc119264676)

[**Кейфреймы** 25](#_Toc119264677)

[**Анимации** 26](#_Toc119264678)

[**Кроссбраузерность** 27](#_Toc119264679)

[**display:flex** 28](#_Toc119264680)

[**Вложенные флекс-контейнеры** 28](#_Toc119264681)

[**Направление внутри флекс-контейнера** 29](#_Toc119264682)

[**Управление положением содержимого флекс-контейнера** 29](#_Toc119264683)

[**Выравнивание флекс-элементов в контейнере** 29](#_Toc119264684)

[**Перенос элементов при сжатии окна браузера** 29](#_Toc119264685)

[**Свойства флекс-элементов. Порядок** 30](#_Toc119264686)

[**Свойства флекс-элементов. Выравнивание отдельного элемента** 30](#_Toc119264687)

[**Свойства флекс-элементов. Базовая ширина элемента** 30](#_Toc119264688)

[**display: grid** 30](#_Toc119264689)

[**Grid** 30](#_Toc119264690)

[**Родительские свойства grid** 32](#_Toc119264691)

[**JS** 35](#_Toc119264692)

[**Ништяки** 35](#_Toc119264693)

[**Переменные** 35](#_Toc119264694)

[**Массив** 35](#_Toc119264695)

[**Операторы** 36](#_Toc119264696)

[**Условные операторы** 37](#_Toc119264697)

[**Тернарные операторы** 37](#_Toc119264698)

[**Несколько операторов «?»** 37](#_Toc119264699)

[**Оператор объединения с null ??** 38](#_Toc119264700)

[**Switch Case** 38](#_Toc119264701)

[**Функции** 38](#_Toc119264702)

[**Строки** 38](#_Toc119264703)

[**Числа** 39](#_Toc119264704)

[**Объект Math** 39](#_Toc119264705)

[**Объекты** 39](#_Toc119264706)

[**JSON** 39](#_Toc119264707)

[**Дата и время** 39](#_Toc119264708)

[**Циклы** 40](#_Toc119264709)

[**Прототипы** 40](#_Toc119264710)

[**DOM** 40](#_Toc119264711)

[**Доступ к элементам HTML** 41](#_Toc119264712)

[**Атрибуты** 41](#_Toc119264713)

[**Классы** 42](#_Toc119264714)

[**Добавление событий** 42](#_Toc119264715)

[**Подключение внешних библиотек** 42](#_Toc119264716)

[**Всплытие и погружение** 43](#_Toc119264717)

[**Делегирование событий** 44](#_Toc119264718)

[**Взаимодейсвие с пользователем** 44](#_Toc119264719)

[**Хранение данных в localStorage** 44](#_Toc119264720)

[**React** 45](#_Toc119264721)

[**Старт** 45](#_Toc119264722)

[**Компонента** 45](#_Toc119264723)

[**Стрелочная функция** 45](#_Toc119264724)

[**Импорта и экспорта** 46](#_Toc119264725)

[**CSS Modules** 46](#_Toc119264726)

[**props** 47](#_Toc119264727)

[**route, browser-router, маршрутизация** 47](#_Toc119264728)

[**NavLink** 48](#_Toc119264729)

[**Функция map** 48](#_Toc119264730)

[**GIT** 48](#_Toc119264731)

[**Консоль** 48](#_Toc119264732)

[**Настройка Git** 48](#_Toc119264733)

[**Подключаем Git к проекту** 49](#_Toc119264734)

[**Работать в командной строке быстро** 49](#_Toc119264735)

[**Оформление репозитория. README.md** 49](#_Toc119264736)

# **Методология БЭМ**

БЭМ была разработана в Яндексе, а сейчас стала стандартом рынка разработки, в том числе зарубежного. В основе методологии лежат два набора правил: для классов и для файлов.

**Правила именования классов.** Следования этим правилам как раз и делают код замодокументируемым.

**Правила организации файлов.** БЭМ определяет, как разбить код на файлы и как эти файлы организовать.

**БЭМ** — аббревиатура от «блок, элемент и модификатор». Именно с этими тремя сущностями оперирует БЭМ.

Блок — это независимый фрагмент страницы: шапка сайта, логотип, строка поиска, кнопка. Это не техническое понятие, а логическое. В официальной спецификации БЭМ блок описан как «логически и функционально независимый компонент страницы».

**Название класса блока** — слово или несколько слов, разделённых дефисом. Например, header и header-block — корректные названия.

**Элемент** — это часть блока, которая не может существовать отдельно от него. Так кнопка «Отправить» — это часть веб-формы, вне формы она не используется. Элемент не может использоваться вне своего блока-родителя. БЭМ рекомендует именовать классы так, чтобы сразу было ясно, что это за компонент — элемент или блок.

**Элементы именуют так:** сначала пишут название блока, к которому относится элемент, потом два нижних подчёркивания \_\_, и затем — имя элемента. Если название элемента состоит из двух слов, вместо пробелов ставят дефис. Правильное имя для ссылки в шапке — **header\_\_link**

**Модификатор** — это свойство блока или элемента, которое переопределяет их параметры. Например, существует блок logo. Он используется в двух местах: в шапке сайта и в подвале. В обоих случаях он выглядит и ведёт себя одинаково, но в шапке логотип чёрный, а в подвале — белый. В этой ситуации применяется модификатор, который изменит только цвет логотипа.

## **Группировка файлов**

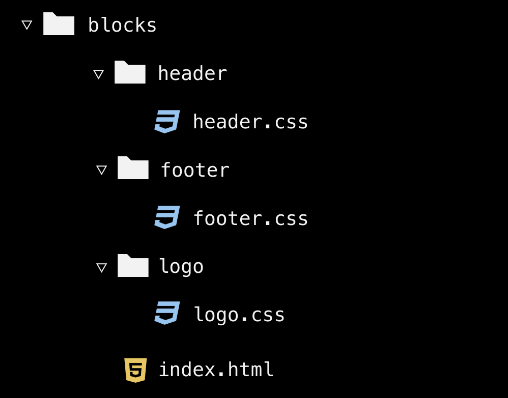
Группировка файлов по методологии БЭМ — объединение файлов не только по их типам, но и по функциональности.

## **Структура для блоков**

Блок в методологии БЭМ — основной строительный материал. Как из кирпичей строят здание, так и из блоков, организованных в сетку, делают веб-страницы. Отсюда требования к коду блоков:

* Код должен быть обособленным — не зависеть от своего окружения. Значит, нам не подходят внешние отступы и позиционирование. Мы стремимся к возможности взять блок и без изменений скопировать его код на новую страницу или в новую зону сайта. Блок должен встать «как влитой», чтобы вам приходилось менять его как можно меньше.
* Называйте блоки понятно. Название блока должно сообщать о его функциональности. Идеально, когда для идентификации блока вообще не нужно отвлекаться от кода и смотреть на страницу.
* Все элементы и модификаторы должны быть внутри блока — без него они не имеют смысла.

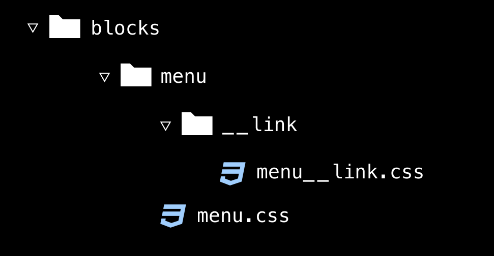
Когда вы продумали структуру блоков на странице, смело создавайте отдельные директории под каждый блок внутри отдельной папки blocks. А в каждой папке блока — CSS-файл с его именем. Эту папку вы будете переносить из проекта в проект; весь код блока будет уже в ней.



## **Структура для элементов**

Элементы в БЭМ — составляющие блока, они не существуют в отрыве от него. Поэтому в БЭМ-структуре весь код элементов размещают внутри папок блоков. Чтобы экономить время, папки называют сокращенно: **\_\_имя-элемента**, ведь из структуры следует, к какому блоку относится элемент.

Имя CSS-файла для элемента пишут полностью, чтобы в коде не приходилось догадываться о принадлежности элемента к блоку, анализируя весь путь до него.



Построив такую файловую структуру, нужно сделать, чтобы страница «увидела» использующийся на ней элемент блока. Для этого в CSS-файл страницы импортируют CSS-файл элемента.

Код index.css:

@import url(../blocks/menu/menu.css); */\* подключаем стили блока \*/*

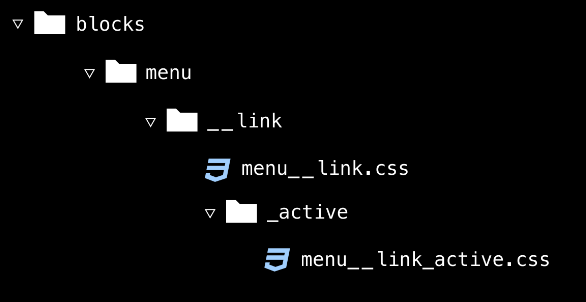
@import url(../blocks/menu/\_\_link/menu\_\_link.css); */\* после блока подключаем стили элемента \*/*

## **Структура для модификаторов**

Модификаторы бывают разные, поэтому вариантов их интеграции в структуру несколько. Модификатор может изменять и блок, и элемент. Папку для файла с модификатором создают внутри той директории, на которую он воздействует. Если модифицирован блок, папка лежит в папке блока, если элемент — в папке элемента. Если модифицируют блок: *Расположение модификатора блока***button**

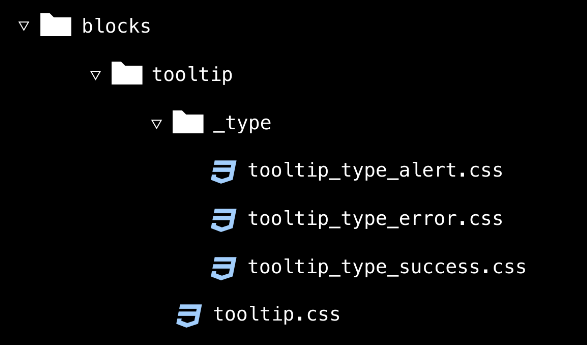


Если модифицируют элемент: *Расположение модификатора элемента***\_\_link**



Как и в случае элемента, папки для модификаторов записывают короткой записью, не повторяя имя блока, а сам файл называют полным именем класса модификатора. Модификаторы в классическом БЭМ-именовании обозначены одним нижним подчёркиванием.

Чаще используют сложные модификаторы вида ключ-значение. Такие модификаторы оформляют в структуре иначе: создают отдельную папку только для ключа, а файлы со значениями вкладывают внутрь. *Модификаторы ключ-значение располагаются в одной директории с названием ключа*



# **Теория**

**Browser** – программа задача которой работать с HTML разметкой с JS и тд. Исп. движек v8

**NodeJS –** Работает с файлами с бд и тд., называются серверным js, принимает и отправляет http запросы.Исп. движек v8.

**npm** – nodejs packeges manager. Пакетный менеджер nodejs.

**SPA** – Single page application. Одностраничная страница. Загружают страницу где мало HTML и много JS единожды, отправляя AJAX запросы и получая в ответ JSON от сервера. Не путать с лендинг страницей

# **HTML & CSS**

[**https://necolas.github.io/normalize.css/**](https://necolas.github.io/normalize.css/) **-** Сброс стилей.

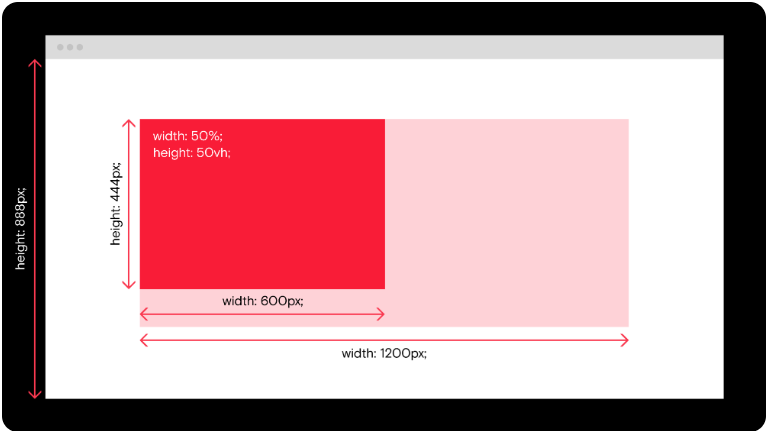
**HTML** – Hyper Text Markup Language, язык гипертекстовой разметки.

**CSS** - Cascading Style Sheets, каскадные таблицы стилей.

**body** – ширина по умолчанию равна видимой ширине окна браузера

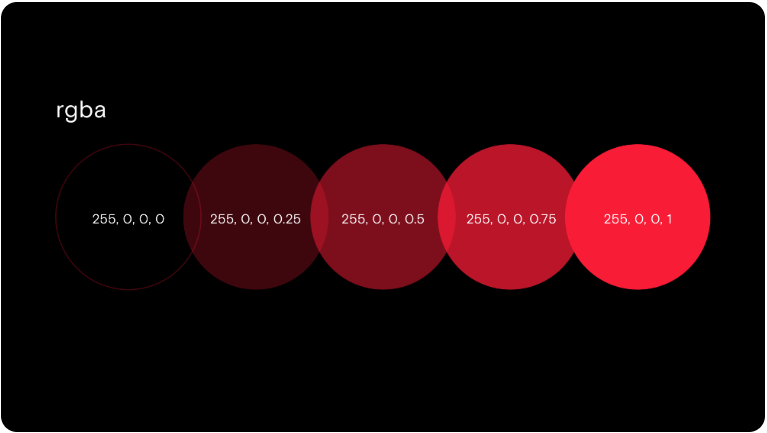
**h1-h6, p, div, section, header\** – по умолчанию занимают 100% ширины своего родителя

**viewport** – окно просмотра в браузере, его базовые характеристики: 100vw & 100vh



Например, если установить элементу свойство height: 50vh, его высота станет равна половине высоты окна просмотра; 100vh — всей высоте, 40vh — 40% высоты. *Что бы при 100vh, у экрана не было скрола, нужно сбросить базовый* ***margin****.*

**Прозрачность** *-* Величина прозрачности меняется от 1 до 0, единица это 100*%. Принято использовать rgba(red green blue alpha).*



## **Описание сайта для браузеров и поисковых машин**

**<meta charset="utf-8">** — определяет кодировку страницы;

**<title></title>** — задаёт имя во вкладке браузера;

**<link rel="shortcut icon" href="favicon.ico">** — задаёт изображение во вкладке браузера.

Сайт могут открывать с разных устройств: компьютеров, планшетов или телефонов. Все эти устройства обладают разными экранами и разрешением. Такое обилие устройств с различными разрешениями экрана вынуждает нас следить за поведением страницы на разных экранах. Поэтому важно прописывать метатег **<meta name="viewport" content="">**. Этот метатег определяет поведение страницы в зависимости от размера дисплея.

**viewport** сообщает браузеру, как выглядит видимая пользователю область страницы. Если этот метатег не указывать, сайт может отображаться по-разному на разных устройствах. Где-то всё будет работать хорошо, а где-то — нет.

**<meta name="viewport" content="width=device-width, initial-scale=1">**

Здесь две инструкции:

1. **width=device-width** означает «занимать всю ширину экрана»;
2. **initial-scale=1** — что страница на любом устройстве должна отображаться в масштабе 1:1, то есть не масштабироваться при загрузке.

## **Метатеги для поисковиков**

Также есть метатеги для поисковых машин: вроде Гугла и Яндекса. Алгоритмы поиска опираются на значения определённых метатегов. Поэтому, чтобы сайт был показан именно тем пользователям, которые им интересуются, следует эти метатеги прописывать.

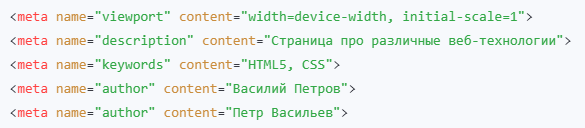
Есть три метатега для поисковиков: описание сайта, его авторы и ключевые слова.

Описание контента прописывают в метатеге **description**. Он содержит небольшой текст — около абзаца — рассказывающий, чему посвящён сайт.

Автора или авторов сайта указывают в метатегах author. Обратите внимание: в один метатег **meta name="author"** вписывается имя одного автора.

Также есть метатег **keywords** — в нём прописывают ключевые слова сайта.

Значение атрибута **content** в этих метатегах указывается в свободной форме на основном языке сайта:



## **Подключение шрифтов к странице**

Шрифт подключают в два этапа:

1. декларация шрифта,
2. применение шрифта элементу.

Сначала нужно указать браузеру, где брать шрифт и что с ним делать. Это и есть декларация. Её выполняют директивой **@font-face** и обычно располагают в начале CSS-файла:

@font-face {

src: url(путь к файлу);

}

Теперь надо дать шрифту имя. За это отвечает свойство font-family:

@font-face {

src: url(путь к файлу);

font-family: 'Best Font Ever';

}

При подключении шрифта к странице может возникнуть проблема: файл шрифта «потеряется» по пути от сервера, или сервер со шрифтом будет недоступен. На такие случаи принято указывать альтернативные шрифты, которые применятся к элементу.

div {

*/\* Если шрифт из файла не загрузился, использовать Arial или Helvetica, на крайний случай — любой шрифт без засечек в системе пользователя \*/*

font-family: 'Best Font Ever', 'Arial', 'Helvetica', sans-serif;

font-weight: bold;

font-style: italic;

}

Шрифты бывают разных форматов.

Чтобы шрифты работали во всех современных браузерах, достаточно подключить шрифты в форматах woff2 и woff. Они есть в любом архиве шрифтов для веба. Старые браузеры не умеют работать с woff2 и woff, поэтому нужно подключать альтернативные шрифты. Чтобы создать список шрифтов в разных форматах, используют параметр format. Браузер прочитает такой список и выберет первый найденный шрифт, с которым умеет работать.

@font-face {

src: url(путь к файлу.woff2) format('woff2'),

url(путь к файлу.woff) format('woff'),

url(путь к файлу.ttf) format('truetype'),

url(путь к файлу.eot) format('eot');

}

## **Разнообразие контейнеров HTML5**

В стандарте HTML5 есть целое семейство контейнеров — семантических тегов.

**Семантические теги** размечают страницу с учётом смысла контента. По ним браузер отличает контент, «шапку» и «подвал» (колонтитулы), навигацию.

Если применять семантические теги:

* поисковым машинам будет проще прочитать и понять содержимое сайта,
* страница станет доступной для слабовидящих,
* код будет понятнее другим разработчикам.

Самые распространённые семантические контейнеры:

**<header></header>** — шапка сайта;

**<footer></footer>** — подвал;

**<nav></nav>** — блок меню (от англ. navigation, навигация);

**<main></main>** — основное содержимое страницы; на многостраничных сайтах этот контейнер содержит всё, чем данная страница отличается от остальных. Имя тега по-английски означает «главный».

**<section></section>** — организует несколько блоков в близкие по смыслу секции.

## **Списки**

В HTML есть два вида списков: маркированный и нумерованный.

**Маркированный список** используют, когда порядок элементов не важен: список покупок, чеклист. По умолчанию такой список оформлен буллитами — маркерами в виде точек. **<ul>…</ul>**

**Нумерованный список** применяют, когда порядок имеет значение: в инструкции, описании процесса. Такой список маркирован цифрами, буквами или их комбинациями **<ol>…</ol>**

**list-style** – управление стилями списка

## **Тег статьи**

Для разметки статьи используют контейнер **article**. Он позволяет поисковым системам распознавать на странице блок, отведённый под статью. **<article>…</ article>**

## **Цитаты**

Цитаты в тексте располагают между тегами **<blockquote></blockquote>**.

У этого тега есть необязательный, но полезный атрибут **cite**. Он позволяет указать автора цитаты.

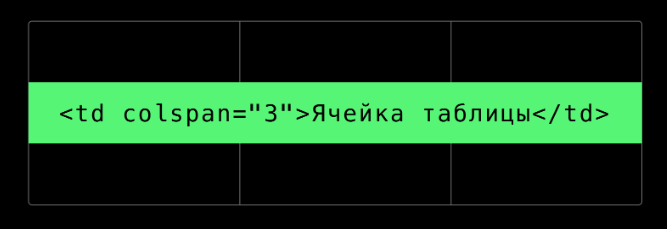
**<blockquote cite="https://praktikum.yandex.ru">Достаточно</blockquote>**

## **Тег <table>**

**<table></table>** - оформить таблицу

Таблица состоит из строк. Их задают тегами **<tr></tr>**. Строки разбивают на ячейки тегами **<td></td>**. Исключение составляют ячейки первой строки, потому что они содержат заголовки колонок. Поэтому такие ячейки задают тегами **<th></th>**.

В самой простой ситуации число ячеек в каждой строке одинаковое. Однако ячейка таблицы может занять место нескольких ячеек:



За это отвечают атрибуты **rowspan** и **colspan**. Запись **<td colspan="3">Ячейка таблицы</td>** означает, что ячейка займёт место трёх столбцов.

## **Тег <form>**

**Форма** — это компонент страницы для получения информации от пользователя. Создается <form></form>

Когда форма объявлена, её наполняют полями. Поля служат для сбора информации: имени пользователя, адреса доставки, согласия на обработку персональных данных и любой другой.

Чтобы по форме было понятно, что и куда вводить, поля для сбора информации объединяют в группы — филдсеты <fieldset >*<получения данных о пользователе>*</fieldset>

По умолчанию fieldset визуально сопровождается серой рамкой. Обычно она не нужна, поэтому её сбрасывают файлом **normalize.css** или CSS-свойством в таблице стилей.

 Форму недостаточно сверстать. Чтобы форма заработала, тег form сопровождают атрибутами. Атрибуты нужны, чтобы указать серверу, чтó ему делать с формой.

## **Поля ввода с тегом input**

Для получения информации от пользователя внутрь формы помещают поля ввода. Большинство из них задаётся тегом input. Причём неважно, что мы ожидаем получить — номер телефона, имя или даже файл — почти всегда подойдёт тег input.

<input type="text"> для текста

<inputtype="number"> для чисел.На мобильном устройстве вместо стандартной будет вызвана клавиатура с цифрами, как на кнопочном телефоне

<inputtype="email"> для электронной почты. Браузер проверяет, есть ли в этом поле точка и @.

<inputtype="password">для паролей. Вместо символов отображаются звездочки или точки.

<inputtype="tel"> для номеров телефонов. Отобразится цифровая клавиатура, как и для number.

<inputtype="range"> создает "ползунок", который можно двигать от максимального до минимального значения.

<inputtype="color"> для интерфейса выбора цвета. В разных браузерах может отображаться по-разному.

<inputtype="url"> для адреса сайта.

<inputtype="date"> дата без времени: 07/03/2009

<inputtype="datetime-local">дата со временем: 04/10/1957 22:28

<inputtype="time"> время: 03:14

Иногда поле нужно заблокировать. Например, пока форма отправляет данные на сервер, стоит запретить возможность редактировать поля. Или при выборе одного из пунктов формы, может стать недоступным выбор другого. Выбрав вечернюю доставку, все утренние временные интервалы следует заблокировать для выбора. Для таких случаев у полей есть атрибут **disabled**. Его прописывают внутри открывающего тега, не указывая значения:

<input type="tel" disabled> Если пользователь предпочитает связь по электронной почте, а не по телефону

Атрибут disabled сделает поле ввода серым: в него нельзя будет установить курсор или выбрать его клавишей Tab.

## **Минимальные и максимальные значения**

Границы диапазона определяются атрибутами **min** и **max**. По умолчанию их значения 0 и 100. Если вам нужны другие границы, определите их вручную: **<input type="range" min="-20" max="500">**

Эти атрибуты работают и с числовыми полями: **<input type="number" min="1970" max="2025">**

Значением ползунков и числовых полей можно управлять клавишами ↑ и ↓. По умолчанию оно будет меняться на 1. Но и это можно переопределить атрибутом **step**:

**<input type="range" min="20" max="50" step="10">**

## **Поля загрузки, сброса и отправки данных**

Тег **input** часто используют для ввода строк и чисел. Но он также может загружать файлы, сбрасывать форму и отправлять её. Это определяет значение атрибута **type**:

<input type="file"> Поле для загрузки файла

<input type="reset"> Кнопка «Сбросить». При нажатии устанавливает все поля формы на значения по умолчанию.

<input type="submit"> Кнопка ”Submit“ (переводится как отправить на рассмотрение). Отправляет данные формы на сервер

<input type="image"> То же, что и submit, но вместо кнопки будет картинка

*Для отправки и сброса формы мы используем тег***input***с атрибутами***submit***и***reset***. Но сейчас всё чаще для этого используют тег***button***:*

<button type="submit">Отправить</button> Отправит форму.

## **Поля ввода с другим синтаксисом**

Не все поля ввода задаются тегом **input**, есть исключения. Тег <input type="text"> подходит для ввода одной строки. Если текст может быть многострочным, лучше применять тег **textarea** (переводится как текстовая область): <textarea></textarea>

Второе исключение из правил тег **select** (англ. выбирать). Он создаёт поле с выпадающим списком, в котором варианты заключены между тегами <option></option> (переводится как вариант):

<select>

<option>Пункт 1</option>

<option>Пункт 2</option>

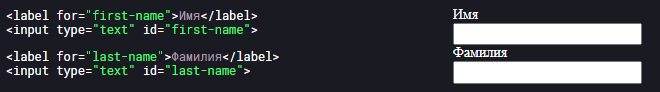
</select>

## **Ярлыки**

<label>Имя</label> - сообщает браузеру и пользователю, что именно вводить в каждое поле.

Указывая, к какому полю относится ярлык, связывают сразу два атрибута.

Первый — атрибут **id** (идентификатор), он задаётся полю ввода. Так элемент получает название, на которое теперь можно ссылаться. За это отвечает атрибут **for** тега **label** (от английского ”for“ в значении «для»). **for** указывает браузеру, к какому полю относится ярлык.



Это один из тех редких случаев, когда в современном **HTML** нужен атрибут **id**. Если связать ярлык и поле ввода описанным образом, при клике на ярлык поле станет активным.

Часто используют ещё один подход к вёрстке ярлыков — поле ввода делают дочерним элементом ярлыка:



В этом случае связывать ярлык с полем ввода необязательно, ведь они уже находятся в родстве. Клик по ярлыку будет активировать поле ввода. Такой подход встречается очень часто, так как решает типичную задачу — расположить ярлык рядом с полем ввода. Но так вы теряете возможность расположить лейбл в другом месте.

## **Передаваемые значения**

После того как пользователь ввёл свои данные в форму, её необходимо передать на сервер. Для этого браузер собирает все данные формы в одну строчку-запрос:



Знак вопроса отделяет адрес сервера от запроса. Тело запроса разделено знаками амперсанд (&) на конструкции вида: **firstname=Аркадий**

Слева от знака равенства — имя поля, его нужно установить атрибутом **name** для каждого поля ввода, иначе данные на сервер не уйдут. <input type="text" name="firstname">

Значения передаёт специальный атрибут value. В него записывается то, что пользователь ввёл в поле. Некоторым полям ввода, например пунктам выпадающих списков и кнопкам отправки и сброса данных в форме, value задают явно:

<input type="text" name="firstname">

<select name="search">

<option value="yandex">Yandex</option>

<option value="some-search">Другой поисковик</option>

</select>

<input type="submit" value="Отправить на сервер">

Атрибут value можно задать и обычным текстовым полям. Эти значения и будут отправлены на сервер, если пользователь их не изменит.

<input type="text" name="firstname" value="**Невведённое имя**">

<select name="search">

<option value="yandex">Yandex</option>

<option value="some-search">Другой поисковик</option>

</select>

<input type="submit" value="Отправить форму сейчас">

## **Поля множественного и единичного выбора radio checkbox**

Поле выбора — это элемент формы со списком, где можно отметить один или несколько вариантов. Такие поля бывают двух типов: с возможностью единичного и множественного выбора.

Оба типа поля создаются тегом **input**, а поведение определяет атрибут **type**.

<input type="radio"> *<!-- поле единичного выбора (радиокнопка) -->*

<input type="checkbox"> *<!-- поле множественного выбора (флажок) -->*

## **Стилизация переключателей и полей выбора**

Необычные поля ввода — такие как **checkbox**, **radio**, **range**, **select** — словом, все те, куда нельзя ввести текст, стилизуются браузером. Через CSS их оформить достаточно сложно, а порой и невозможно. Вот где используют такой приём: поле визуально скрывают, а поверх создают аналог с собственным дизайном. Первое, что нужно — это скрыть текущее поле ввода. Но сделать это так, чтобы оно осталось на странице, ведь мы хотим, чтобы оно активировалось с клавиатуры.

Делаем поле ввода дочерним для label.

Мы будем скрывать основной input и менять на новый. Принципиально важно, чтобы label окружал эти элементы. Мы будем активировать новый элемент по клику на лейбл:

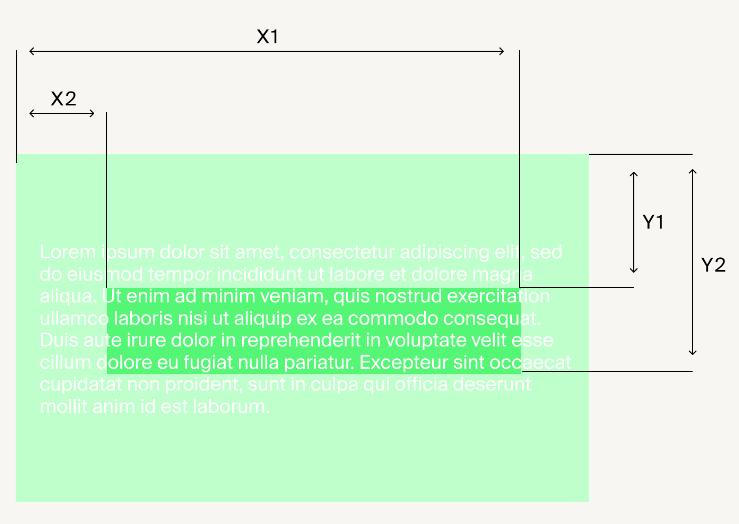
<label>

<input type="checkbox">

</label>

Скрываем элемент через **clip**.

Свойство **clip** работает с абсолютно спозиционированными элементами и отвечает за область, внутри которой будет показано содержимое этого элемента. clip(X1, Y1, X2, Y2);



Если все координаты нулевые, элемент не будет показан. Такой код скроет чекбокс, и это будет кроссбраузерно:

.input[type="checkbox"] {

position: absolute;

width: 1px; */\*если у элемента не будет размеров, некоторые браузеры посчитают, что его вообще нет\*/*

height: 1px;

overflow: hidden; */\* без этого свойства тоже сработает, но так наверняка. Мы его ещё изучим, оно скрывает элементы, выходящие за границы родителя \*/*

clip: rect(0 0 0 0);

}

Когда чекбокс скрыт, нужно создать второй элемент рядом с ним, он будет заменять скрытый новым дизайном. Примерно так:

<label>

<input type="checkbox" class="invisible-checkbox"> *<!-- этот элемент мы скрыли -->*

<span class="visible-checkbox"></span> *<!-- этот будем стилизовать -->*

</label>

Стилизация альтернативного элемента — дело техники. Немного позиционирования, отступов, граница — этого хватит. Важно, чтобы новый элемент расположился там, где вам нужно, и был похож на чекбокс.

Описать все состояния чекбокса через псевдоклассы

Состояния чекбокса:

* включённый и выключенный — определяется псевдоклассом **disabled**;
* выбранный и не выбранный — определяется псевдоклассом **checked**;
* в фокусе и не в фокусе — определяется псевдоклассом **focus**.

Получаются такие селекторы:

input[type="checkbox"]:disabled {} */\* выключенный \*/*

input[type="checkbox"]:checked {} */\* выбранный \*/*

input[type="checkbox"]:focus {} */\* в фокусе \*/*

Есть проблема. Наш чекбокс скрыт, поэтому на состояние должен реагировать альтернативный элемент. Выбрать его и стилизовать — целая задача. Нам потребуется очередной необычный селектор — селектор первого соседа, следующего за нашим скрытым чекбоксом. Этот сосед и станет альтернативным элементом. Селектор первого соседа определяется символом +:

input[type="checkbox"] + span {} */\* так для чекбокса выглядит первый сосед с именем span \*/*

## **Подсказки в текстовых полях placeholder**

**placeholder** - Добавить к текстовым полям подсказки.

## **Стилизация placeholder**

Стили для плейсхолдера прописывают селектором с псевдоэлементом **::placeholder**

input::placeholder {

font-family: Arial;

}

Этот код задаёт семейство шрифтов для всех плейсхолдеров. Но в браузерах компании Microsoft (Internet Explorer и Edge) псевдоэлемент **::placeholder** не поддерживается, и код не сработает. Чтобы стилизовать плейсхолдеры в IE, псевдоэлемент предваряется вендорным префиксом **-ms-input** — **::-ms-input-placeholder:**

input::-ms-input-placeholder {

font-family: Arial;

}

Ранее сам стандарт CSS не позволял стилизовать плейсхолдеры. Поэтому каждый производитель браузера придумал свои вендорные префиксы, чтобы поддержать экспериментальные решения:

input::-webkit-input-placeholder {color:#c0392b;} */\* для Safari и Chrome \*/*

input::-moz-placeholder {color:#c0392b;} */\* для Firefox \*/*

input::-ms-input-placeholder {color:#c0392b;} */\* для IE \*/*

Вот такой код обеспечит железобетонную надёжность работы во всех браузерах:

input::-webkit-input-placeholder {color:#c0392b;}

input::-moz-placeholder {color:#c0392b;}

input::-ms-input-placeholder {color:#c0392b;}

input::placeholder {color:#c0392b;}

## **Обязательные поля required**

Поля формы можно помечать как обязательные. Без заполнения таких полей форму не получится отправить — браузер выдаст сообщение об ошибке.

<input type="text" required>

## **Особенности стилизации текстовых полей ввода, кнопок и ярлыков**

У текстовых полей ввода только одна функция — получить текст.

В них не выбирают варианты и это не переключатели. Вёрстка таких полей довольно интуитивна, можно пользоваться уже знакомыми CSS-стилями. Тем не менее, есть ряд особенностей.

1. Все поля ввода и лейблы по умолчанию строчные элементы. Поэтому вам часто будет нужно свойство display, чтобы управлять их расположением.
2. Поля ввода не наследуют некоторые стили от своих родителей. Например, стили шрифтов. Какой бы шрифт ни был задан форме, в полях ввода пользователь будет всё набирать стандартным для браузера шрифтом. Поэтому всем полям ввода нужно задавать наследование явно. Для этого существует специальное значение inherit (унаследовать). Оно означает приказ «унаследовать от родителя»: input {font-family: inherit; */\* Теперь поле ввода наследует шрифт от родителя \*/*

}

1. Когда пользователь выбрал поле, нужна возможность управлять его поведением. Например, задать активному полю цвет рамки. Для этого есть специальный псевдокласс **:focus:**

input:focus {*/\* стили для активного поля \*/*}

По умолчанию в состоянии **:focus** элементы ввода окружены синей внешней линией, или обводкой: Обводка полезна, когда между элементами формы пользователь перемещается не мышью, а нажатием клавиши Tab. Благодаря обводке люди со слабым зрением лучше видят элемент в фокусе. Настраивается эта полезная опция свойствами группы **outline**:

input:focus {

outline-color: yellow; */\* цвет обводки — жёлтый \*/*

outline-style: dashed;*/\* стиль линии — прерывистая \*/*

outline-width: 3px; */\* толщина обводки — 3px \*/*

}

## **Обработчик поля input**

У тега **form** есть два атрибута: **action** (переводится как действие) и **method** (переводится как метод).

Значение **action** — это ссылка на обработчик. Этим атрибутом мы сообщаем, какой программе отдавать данные.

Атрибут **method** принимает значения **GET** или **POST**. Метод **GET** (получить) нужен, когда информация уже есть на сервере и вам нужно получить её себе в браузер. Метод **POST** (отправить) — когда данные есть у вас в браузере и их надо обновить на сервере.

<form action="ссылка на обработчик" method="POST">

## **iframe**

iframe — сокращение от inline frame, что значит «встроенная в строку рамка». Это контейнер, в который помещают информацию с других сайтов. По сути — обычный элемент страницы. Можно задать ему ширину и высоту: в рамке таких размеров и будет отображаться подгружаемая информация.

Атрибуты тега iframe:

height - высота фрейма в пикселях

<iframe height="250px"></iframe>

width - ширина в пикселях

<iframe width="250px"></iframe>

src - ссылка на веб-страницу, откуда поступают данные

<iframe src="https://music.yandex.ru/iframe/#track/41284288/5400781/"></iframe>

allowfullscreen - разрешает полноэкранный режим

<iframe width="100%" allowfullscreen src="https://music.yandex.ru/iframe/#track/41284288/5400781/"></iframe>

frameborder — это устаревший атрибут, он отображает вокруг содержимого фрейма рамку с эффектом трёхмерности. По умолчанию рамка рисуется и значение этого атрибута frameborder="1". Значение "0" её скрывает.

*В HTML5 не применяют этот атрибут, а управляют рамкой через свойство border в CSS. Но frameborder встречается в популярных готовых фреймах, к примеру — в коде вставки видео с Youtube. Вам нужно знать этот атрибут, чтобы найти его и удалить, иначе валидатор выдаст ошибку.*

<iframe frameborder="0" width="600" height="100" src="https://music.yandex.ru/iframe/#track/41284288/5400781/"></iframe>

## **Тег <video>**

<video> - Для размещения видео на сайте. <video src="url" >

Сейчас тег работает с четырьмя форматами: MP4, MOV, WebM и OGG. Форматов несколько, потому что нет единого формата, который поддерживался бы всеми браузерами.  Если поддержки в нужном браузере нет, можно задействовать дополнительный тег **<source>** (источник). Тогда форматы указывают в значениях атрибута **type**:

<video width="320" height="240" controls>

<source src="kino.mp4" type="video/mp4">

<source src="kino.ogg" type="video/ogg">

*<!-- Если не сработает формат mp4, браузер попробует ogg -->*

</video>

## **Атрибуты тега <video>**

autoplay — видео запускается, как только загрузится.

<video autoplay>

controls — станет доступен виртуальный пульт управления с ползунком изменения громкости, кнопками воспроизведения, паузы, перемотки и прочими инструментами:

<video controls>

loop — как только видео закончится, оно запускается с начала:

<video loop>

muted — звук будет отключён:

<video muted>

src — ссылка на воспроизводимый ролик:

<video src="URL">

type — формат видео:

<video type="video/mp4">

height — высота области воспроизведения в пикселях:

<video height="250px">

width — ширина области воспроизведения в пикселях:

<video width="250px">

poster — ссылка на изображение, которое пользователь видит до того, как нажмёт ”Play“, или пока видео загружается. Если **poster** не указан, то в качестве превью браузер покажет первый кадр видео, когда он загрузится.

<video poster="/images/example.gif">

## **Тег <audio>**

Тег работает с тремя форматами аудио: MP3, WAV и OGG. Если браузер не поддерживает необходимый формат, мы снова применим тег **<source>**

Атрибуты тега audio:

autoplay — аудиозапись исполняется, как только загрузится:

<audio autoplay>

controls — станет доступен виртуальный пульт управления с ползунком изменения громкости, кнопками воспроизведения, паузы, перемотки и прочими инструментами:

<audio controls>

loop — как только аудио закончится, оно заиграет снова:

<audio loop>

muted — режим отключения звука:

<audio muted>

src — ссылка на путь к аудиофайлу:

<audio src="URL">

type — формат аудио:

<audio type="audio/ogg">

## **Селекторы**

*По имени тега:*

**p {}** - выбираем все элементы с тегом p на странице

*По имени класса:*

**.my-class {}** - выбираем все элементы с классом my-class

*Селекторы вложенности:*

**div p {}** - все p внутри div

**div .my-class** - все элементы с классом my-class внутри div

*Комбинированные селекторы:*

**a.my-link {}** - ссылка с классом my-link

**.my-button.my-button\_large {}** - элемент с классами my-button и my-button\_large одновременно

## **Селекторы атрибутов**

В CSS есть селекторы, способные выбирать элементы с любым значением любого атрибута. Вы можете выбрать ссылки, ведущие на определённый адрес, изображения с определённым атрибутом alt и тому подобное. Выглядит это в коде так: любой-селектор[любое-имя-атрибута="любое-значение-атрибута"] {...}

Этот тип селекторов может быть полезен при работе с формами, ведь большинство полей ввода отличаются только атрибутом **type** и работают по-разному в зависимости от него. Например, задать стили для однотипных полей можно, записав после слова **input** селектор атрибута — пару **атрибут="значение"**:

input[type="submit"] {*/\* Стили для кнопки submit \*/*}

## **Селектор первого соседа**

Селектор первого соседа определяется символом +

input[type="checkbox"] + span {} */\* так для чекбокса выглядит первый сосед с именем span \*/*

.form\_\_item\_el\_extra-options:focus + .form\_\_pseudo-item\_type\_checkbox {}*/\* так для класса в фокусе выглядит первый сосед с классом form\_\_pseudo-item\_type\_checkbox* *\*/*

## **Псевдоклассы**

Можно стилизовать элементы, реагируя на действия пользователя. Обычно нужно определить специфический внешний вид элемента:

**link** - если пользователь ещё не посещал ссылку.

**visited** - ссылка, по которой пользователь уже переходил.

**hover** - навёл на элемент указатель мыши.

**active** - элемент, по которому кликают (вид элемента в момент нажатия).

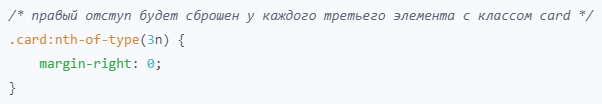
*CSS-селекторы псевдоклассов записывают через двоеточие*

**nth-of-type** - «n-й элемент этого типа». Он назначает стили каждому n-му элементу: например, второму, третьему, десятому или пятидесятому. В скобках после имени псевдокласса указывают **число и букву n**.

**2n** означает, что свойство нужно применить к каждому второму элементу, подходящему под селектор.

**3n** - каждому третьему

**4n** - каждому четвёртому. И так далее по аналогии.



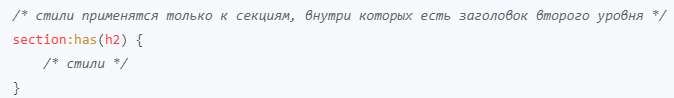
**first-of-type -** стилизовать самый первый элемент.

**last-of-type -** для самого последнего элемента.

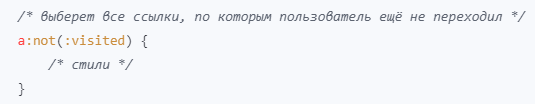
## **Функциональные псевдоклассы**

Эти псевдоклассы позволяют стилизовать элементы, анализируя разметку или историю действий пользователя.

**has** - позволяет стилизовать только элементы, которые содержат внутри себя заданные теги.



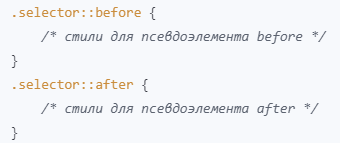
**not** - принимает в скобках, должно не выполняться.



## **Псевдоэлементы и селекторы атрибутов**

Браузер может не только анализировать ваш контент, но и добавлять что-то от себя до или после элемента.

Этому служат псевдоэлементы **before** (англ. «до») и **after** (англ. «после»). Их принято писать через два двоеточия в конце селектора:



Браузер вставляет псевдоэлемент **::before** перед содержимым указанного элемента, а **::after** — после.

Псевдоэлементы **::before** и **::after** всегда содержат свойство content. Оно определяет, каким содержимым будут наполнены псевдоэлементы.



Этот код выведет на страницу заголовок «*Зима недаром злится*». При этом слова «*Зима*» и «*злится*» будут не частью элемента **h1**, а вложенными в **h1** особыми псевдоэлементами, то есть «*элементами, которых как бы нет*».

Отображением псевдоэлементов управляют посредством CSS точно так же, как и отображением обычных элементов: задают им фон, шрифт, размер и прочие стили.

## **Наследственность в CSS**

Стиль родительских элементов оказывает влияние на стиль дочерних.

Например, если абзацу **p** не задан конкретный размер шрифта, цвет текста или выравнивание, абзац будет «прислушиваться» к родителю, то есть к стилю элемента **div**.

И если для **div** определить свойство **font-size: 40px**, высота символов во вложенном элементе p тоже станет **40px**. Установите для **div** выравнивание по центру **text-align: center** — и текст абзаца **p** выровняется по центру. Но если дочернему элементу установить собственное конкретное значение, то свойство наследоваться не будет.

## **Позиционирование элементов.**

Взаимное расположение элементов в макете называется позиционированием. CSS-свойство position определяет, как браузер расположит элемент в потоке.

По умолчанию для всей страницы применяется статическое позиционирование, которое описывается правилом: **position: static;**

Это значит, что элемент:

* находится в потоке; все другие элементы знают, где он расположен, и реагируют на его смещение;
* перемещается визуально только за счёт отступов. Так как отступы — часть блочной модели элемента, то с точки зрения браузера мы не двигаем элемент, а меняем его размеры за счёт отступов.

## **Относительное позиционирование**

**position: relative;** - Когда элемент остаётся в потоке, но смещается относительно самого себя, это называют относительное позиционирование.

Ключевые моменты этого способа позиционирования:

Поскольку элемент остаётся в потоке, отведённое под него пространство видят другие элементы.

Элемент не отступает от соседей, как при статическом позиционировании, а действительно смещается относительно первоначального положения соответствующими инструкциями:

**top** — от верхней границы блока;

**right** — от правой границы;

**bottom** — от нижней границы;

**left** — от левой границы.

Сдвигается только сам элемент, а выделенная под него область в потоке остаётся на месте.

## **Фиксированное позиционирование**

**position: fixed;** - фиксированный вид позиционирования, положение элемента указывается относительно окна браузера.

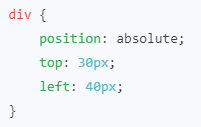
Пространство, отведённое под элемент, займут следующие за ним элементы в потоке. А фиксированный элемент заслонит их, расположившись поверх соседей.

Среди свойств **top, right, left, bottom** существует приоритет: **top** и **left** важнее **bottom** и **right** .

## **Абсолютное позиционирование**

**position: absolute;** - элемент, спозиционированный абсолютно, тоже вырван из потока, то есть его пространство в потоке заняли элементы, которые следовали за ним.

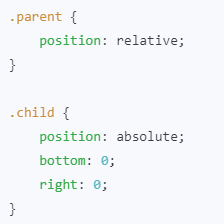
В отличие от фиксированного позиционирования, абсолютно спозиционированный элемент может вести себя по-разному в зависимости от контекста, в котором находится.

**Первый случай** — если его родительский элемент неспозиционирован, имеет заданное по умолчанию свойство **position: static;.** Тогда интересующий нас блок станет искать ближайшего сверху по иерархии прародителя, чтобы спозиционироваться относительно его границ и если не найдёт, расположится относительно границ **body**. То есть в неспозиционированном окружении код: расположит блок на 30 пикселей ниже верхней границы body и на 40 пикселей правее левой.

В отличие от фиксированного позиционирования, элемент не останется на месте при прокрутке, так как он расположен относительно тела документа, а не окна браузера.

**Второй случай:** родительский элемент спозиционирован, то есть имеет значение свойства **position**, отличное от **static**. Тогда наш спозиционированный абсолютно блок будет размещён согласно инструкциям **top, right, bottom, left** относительно границ родителя.

Если **.parent** — родительский элемент для **.child**, такой CSS-код расположит блок **.child**в правом нижнем углу блока **.parent:**



## **z-index**

Для управления близостью элементов используется свойство **z-index**. Его название говорит о том, что задействовано третье измерение. Позиционирование идёт уже не по ширине (ось **x**) и не по высоте (ось **y**), а по глубине, т.е. по оси **z**.

По умолчанию у всех элементов одинаковый **z-index**, равный 0.

**z-index** не действует на элемент со свойством **position: static** — или если свойство **position** вообще не задано. Из этого правила есть исключение: если элемент находится во **flex**- или **grid**-контейнере (у родителя задано **display: flex** или **display: grid**), то **z-index** сработает даже когда позиция элемента — **static**.

## **Разделение файлов по типам**

**Файловая структура** — это способ организации файлов по каталогам.

Удобнее разделять файлы по папкам, отводя для изображений, шрифтов, файлов стилей и скриптов свои каталоги. Это облегчает поиск. Файловая структура со вложенными директориями называется иерархической.

Вот файловая структура, которую часто организуют веб-разработчики:

* HTML-файлы хранятся в корневом каталоге.
* В этот же «корень» вложены подкаталоги —
* /images, изображения — иконки, фоновые и контентные изображения различных расширений;
* /styles, файлы стилей;
* /scripts, JS-файлы;
* /fonts, шрифты.

## **CSS-директивы. @import**

Директивы — инструкции для CSS, которые меняют отображение или поведение элементов на странице. Ими можно подключать CSS-код и шрифты, создавать анимации. Обязательная часть директивы состоит из коммерческой at @ и названия директивы: **@имя\_директивы** ***правило;***

**@import url(main.css);**

У **@import** есть ряд особенностей:

* CSS работает с @import так, что каждая обрабатываемая директива загружается по очереди. Это может плохо отразиться на скорости загрузки сайта. Элемент <link> позволяет загружать файлы параллельно.
* @import будет работать, только когда написана в начале файла. Внешние файлы загружаются первыми. Следующие за ними стили могут их перезаписать.

## **Внешний и внутренний фокус элемента**

Если задавать ширину диву, и добавлять внутрение отступы, то ширина элемента увеличивается. Т.е. базово применяется правило **box-sizing: content-box.**

**box-sizing: border-box** изменяет фокус элемента, границы и внутрений отступ отрисовываются внутрь, и общая ширина будет равна значение **width**.

## **Блочные + строчные**

**Блочный элемент** занимает по умолчанию всю ширину родительского элемента.

**Строчные элементы** занимают ровно столько места, сколько контента в них содержится. Если они идут подряд, то по умолчанию все расположены на одной строке. Им невозможно задать ширину или высоту — они игнорируют указание размеров через стили.

Что делать, когда блоки с определенными размерами должны следовать друг за другом по горизонтали и не занимать всю строку? Можно задать элементам комбинированный тип — блочно-строчный. С одной стороны, они не занимают собой всю горизонталь, с другой, восприимчивы к указанию размеров через CSS. Например, так ведут себя элементы **img**.

Переопределяется тип свойством **display**:

**display: block;** сделает элемент блочным

**display: inline;** сделает элемент строчным

**display: inline-block;** сделает элемент блочно-строчным, элемент ведет себя как строчный, но не игнорирует свойства **width** & **height**.

*Обнулить отступ* - **font-size: 0** у родительского класса, у дочерних элементов применен **display: inline-block;**

## **Свойства фона**

**background-color** - цвет фона. Можно задать словами, HEX-кодом, в RGB- или RGBA-палитре.

**background-image -** фоновое изображение. Ссылка на ресурс задаётся внутри параметра **url()**

**background-size** - размер фона. Часто используют значения:

**contain** — фоновое изображение целиком вписывается в блок;

**cover** — фон накрывает блок.

**background-position** - расположение фона по вертикали и горизонтали

## **Расположение элементов по центру: margin: auto;**

Можно на глаз задать количество пикселей для **margin-left** первой карточки, пока правила не окажутся в центре. Но такой способ сработает только на вашем компьютере, а для экрана другого размера потребуется другое число пикселей. С центрированием блочных элементов работает особое значение **auto**. Оно автоматически устанавливает максимально возможный отступ по горизонтали. Если задать свойствам **margin-left** и **margin-right** значение **auto**, получатся максимальные отступы с обеих сторон, и элемент встанет по центру своего родителя.

## **Центрировать по вертикали**

Что бы центрировать элемент по вертикале, элементу нужно добавить родителю **display: flex;**

**margin: auto** у дочерних элементов с **flex** свойством отцентровывается не только по горизонтале, но и вертикале.

## **Тени**

**box-shadow** – создание свойства тени

**text-shadow** – создание свойств тени тексту

Последовательность значений, которые описывают тень(box-shadow: -2px 2px 5px #FD6969):

* горизонтальный сдвиг (отрицательное значение помещает тень левее элемента, положительное — правее);
* вертикальный сдвиг (отрицательное значение помещает тень выше элемента, положительное — ниже);
* радиус размытия (чем значение больше, тем тень шире и бледней);
* цвет (задаётся так же, как цвет текста или фона).

Иногда тень не нужно сдвигать, достаточно только размытия. В этом случае указывают, что первые два значения равны 0.

Тени — полезный инструмент дизайна. Их можно применять к блочным элементам и текстам. За тень блока отвечает свойство **box-shadow** (от англ. box shadow, «тень блока»), а за тень текста — **text-shadow**(от англ. text shadow, «тень текста»).

Последовательность значений, которые описывают тень, для текстов и блоков одинакова:

* горизонтальный сдвиг тени (отрицательное значение создаёт тень на таком-то расстоянии слева, положительное справа);
* вертикальный сдвиг тени (отрицательное значение создаёт тень на таком-то расстоянии сверху, положительное снизу);
* радиус размытия тени (чем это значение больше, тем тень шире и светлей);
* цвет тени (задаётся так же, как цвет текста или фона).

div {

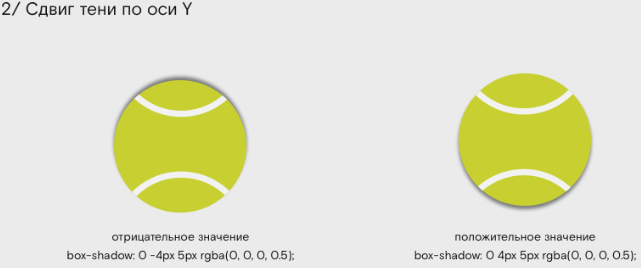
box-shadow: 5px 6px 7px red;

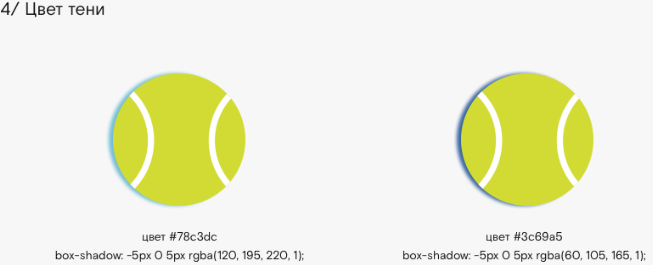
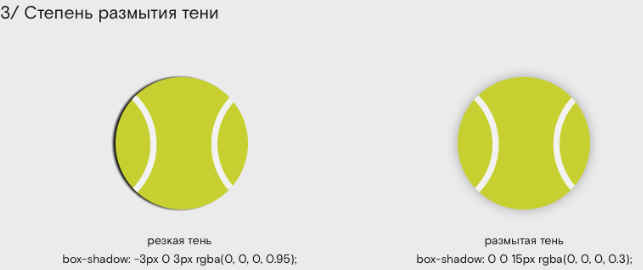
}

h1 {

text-shadow: 5px 6px 7px red;

}





## **Линейный градиент**

Градиент — это вид заливки блока цветом, когда контрольные цвета фиксируют в определённых точках, между которыми браузер создаёт плавные переходы.

Определяется в коде через значение **linear-gradient()** для **background-image.**

В самом простом случае указывают в скобках через запятую два цвета - начальный и конечный. Браузер сделает между ними плавный переход.

background-image: linear-gradient(#6be454, #57d9ba)

По умолчанию браузер отрисовывает линейный градиент сверху вниз. Это поведение можно изменить. Один из вариантов — задать поворот градиента в градусах.

Например такой код повернёт градиент на 90 градусов.:

div {

background-image: linear-gradient(90deg, #0078FF, #C2E3E3);

}

Для построения более сложных градиентов можно использовать более одного цвета:

div {

background-image: linear-gradient(#0B2337, #126DDC, #76C2E0, #D1DC9D, #F09174);

}

После указания каждого цвета можно задать интервал, в котором нужно применить цвет. Первое значение — начальная точка указанного направления, второе — финальная точка.

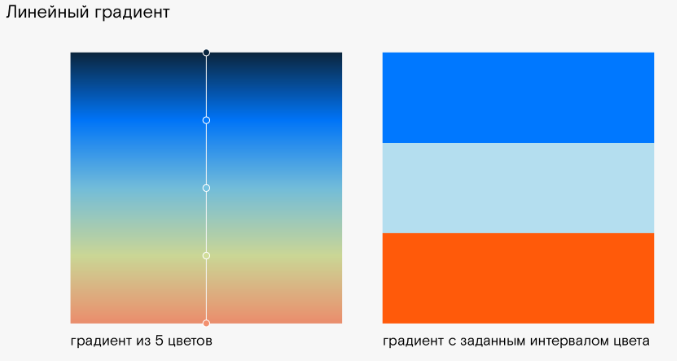
div {

width: 300px;

height: 300px;

background-image: linear-gradient(#0078FF 0px 100px, #B4DEEF 100px 200px, #FF5A0A 200px 300px);

}



Можно также определять точки перелома в процентах:

div:first-of-type {

width: 300px;

height: 300px;

background-image: linear-gradient(#0078FF 20%, #C2E3E3 100%);

}

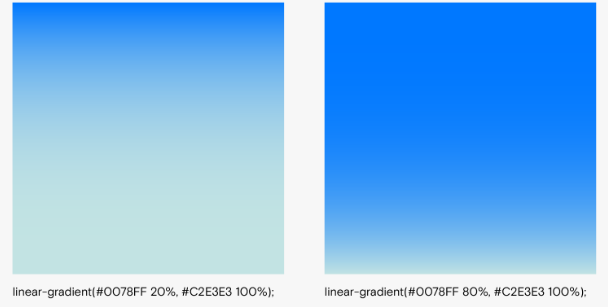
div:last-of-type {

width: 300px;

height: 300px;

background-image: linear-gradient(#0078FF 80%, #C2E3E3 100%);

}



## **Радиальный градиент**

Если цвета линейных градиентов меняются вдоль линии, задающей направление, то при радиальном градиенте изменения цвета расходятся от заданного центра, а сам градиент может принимать форму круга или эллипса.

Самый простой радиальный градиент задаётся значением **radial-gradient()** так:

div {

background-image: radial-gradient(#0078FF, #C2E3E3);

}

По умолчанию центр градиента находится в середине блока, но его можно сместить, задав позицию ключевым словом **at** (здесь в значении «в такой-то точке»).

div {

width: 300px;

height: 300px;

background-image: radial-gradient(at 40px 50px, #0078FF, #C2E3E3);

}

Такой код поместит центр градиента на 40 пикселей правее левого края блока и на 50 пикселей ниже верхнего края.

Как с линейным градиентом, можно определять несколько цветов и точки перелома в процентах:

div {

width: 300px;

height: 300px;

background-image: radial-gradient(#002918 10%, #B4DEEF 20%, #FFFFFF 50%, #FF5831 100%);

}

или в пикселях:

div {

width: 300px;

height: 300px;

background-image: radial-gradient(#002918 50px, #B4DEEF 70px, #FFFFFF 100px, #FF5831 200px);

}

Определение точных интервалов тоже доступно:

div {

width: 300px;

height: 300px;

background-image: radial-gradient(#002918 0px 20px, #B4DEEF 20px 40px, #FFFFFF 40px 60px, #FF5831 60px 80px);

}



Если вы задаёте ширину и высоту первыми параметрами в скобках, то эллипс растянется или сожмётся:

div {

width: 300px;

height: 300px;

background-image: radial-gradient(40px 100px, #0078FF, #C2E3E3);

}

## **Переполнение блоков**

Что происходит, если попробовать поместить блок размерами 500x500 в блок размером 300x600?

Больший блок вышел за рамки меньшего, и это будет видно. Такая ситуация называется в CSS переполнением, а поведение родительского блока при переполнении можно определить свойством **overflow** (англ. overflow, «переполнение»). У него есть четыре возможных значения:

* **visible** — значение по умолчанию,
* **auto** — скрывать зоны, которые выходят за границы блока, но показывать скролл (полосу прокрутки),
* **scroll** — всегда показывать скролл, если его не прячет браузер
* **hidden** — скрыть не поместившуюся часть целиком без скролла.

У свойства overflow есть подтипы с теми же значениями:

* **overflow-x** — определяет поведение по горизонтали,
* **overflow-y** — определяет поведение по вертикали.

У этих свойств есть особенность: добавив одному из свойств значение, отличное от visible, оно перестает работать у другого.

## **Трансформации**

Трансформации — это способ изменения формы элементов. В CSS за трансформации отвечает свойство **transform.**

Любую трансформацию можно описать значением **matrix** c параметрами — матрицей трансформаций, или матрицей преобразований.

div {

transform: matrix(2, 3, 4, 5, 6, 7);

}

Вот что означает каждый из шести параметров:

2 — изменение масштаба по горизонтали (элемент увеличен вдвое по оси X);

3 — скос по вертикали (положительное значение поднимает левую половину элемента и опускает правую, отрицательное наоборот);

4 — скос по горизонтали (положительное значение наклоняет элемент влево, отрицательное вправо);

5 — изменение масштаба по вертикали (элемент растянут в 5 раз по оси Y);

6 — сдвиг по оси X в пикселях;

7 — сдвиг по оси Y в пикселях.

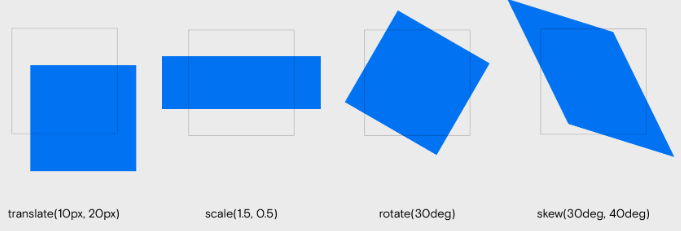
Матрицей можно описать практически любое изменение формы объекта, но для самых популярных есть специальные обозначения.

transform: translate(10px, 20px); */\* сдвинет объект на 10 пикселей вправо и на 20 пикселей вниз \*/*

transform: scale(1.5, 0.5); */\* увеличит объект в полтора раза по горизонтали и уменьшит наполовину по вертикали \*/*

transform: rotate(30deg); */\* повернёт объект на 30 градусов по часовой стрелке \*/*

transform: skew(30deg, 40deg); */\* наклонит объект на 30 градусов влево и скосит левую сторону на 40 градусов вверх, а правую на 40 градусов вниз \*/*



Сдвиги **translate** заданы в пикселях, но возможно определять их и в относительных величинах.

Масштабирование **scale** определяется относительно 1. Значение 1 означает «один к одному», то есть элемент не изменится. Числа больше единицы увеличивают объект, а меньше единицы — уменьшают.

Повороты **rotate** и скосы **skew** можно определять в градусах **deg** (от англ. degree, обычный градус, каких в окружности 360). Допустимы отрицательные значения, чтобы реализовать повороты против часовой стрелки и скосы вправо.

.demo\_\_item\_character\_starman {

background-image: url(https://pictures.s3.yandex.net/animation\_topic/starman.png);

transition: box-shadow .3s ease;

}

## **Плавность переходов**

Эффекты перехода устанавливает универсальное свойство **transition.** У него есть особые значения:

* **transition-property** (англ. property, «свойство») — сообщает, какие именно свойства должны меняться плавно;
* **transition-duration** (англ. duration, «продолжительность») — время перехода в секундах;
* **transition-timing-function** (англ. timing function, «функция отсчёта времени») — описывает, как будет вести себя анимация (она может ускоряться, замедляться или двигаться линейно);
* **transition-delay** (англ. delay, «отсрочка») — задержка в секундах между изменением состояния элемента и запуском анимации.

Все эти значения можно задавать по отдельности:

div {

background-color: black;

transition-property: background-color;

transition-duration: 1s;

transition-timing-function: linear;

transition-delay: 2s;

}

div:hover {

background-color: white;

}

Или сокращённой записью:

div {

background-color: black;

transition: background-color 1s linear 2s;

}

div:hover {

background-color: white;

}

**timing-function** можно задать ключевыми словами или в формате cubic-bezier.

Вот самые популярные варианты, которые можно задать ключевыми словами:

ease (непринуждённо)

* **linear** (линейно)
* **ease** (непринуждённо)
* **ease-in** (осторожно)
* **ease-out** (отпустить)
* **ease-in-out** (сначала осторожно, потом отпустить)

div {

transition: transform .4s ease-out;

}

## **Кейфреймы**

Кейфреймы объявляются директивой **@keyframes** — все директивы пишут после знака **@**.

Объявление (декларация) должно обязательно содержать имя. По нему мы будем вызывать анимацию. Имя следует прямо за директивой через пробел:

**@keyframes name\_of\_keyframe** (здесь name\_of\_keyframe — от английского выражения «имя кейфрейма», в напоминание, что имена лучше всего давать английские и базовой латиницей).

После декларации открывают фигурные скобки «тела кейфреймов».

@keyframes name\_of\_keyframe {

<здесь будут правила изменения стилей>

}

В фигурных скобках описаны **CSS-свойства** элемента в каждом ключевом кадре. Кейфремы при этом указывают в процентах. **0%** соответствует началу анимации, **100%** — концу, **50%** — середине и так далее.

@keyframes move {

0% {

left: 0;

top: 0;

}

25% {

left: 30px;

top: 30px;

}

50% {

left: 60px;

top: 0;

}

75% {

left: 90px;

top: 30px;

}

100% {

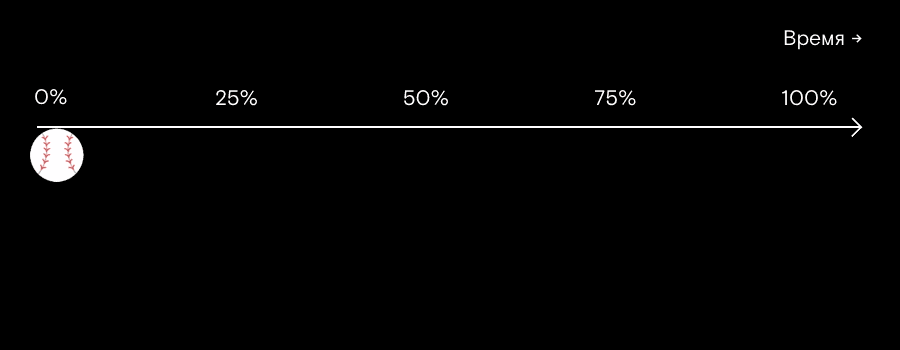
left: 120px;

top: 0;

}

}

Такой код определяет анимацию из пяти ключевых кадров. Каждый кейфрейм задаёт положение объекта. Между этими положениями элемент и будет плавно перемещаться:



Если состояний всего два — начальное и конечное, возможна упрощенная запись.

Так, в этом кейфрейме **rotation** (англ. rotation, «вращение») описан полный оборот от **0°** до **360°**, заданный двумя значениями **rotate** свойства **transform**:

@keyframes rotation {

from {

transform: rotate(0deg);

}

to {

transform: rotate(360deg);

}

}

## **Анимации**

За применение анимации к HTML-элементу отвечает группа свойств **animation**.

Чтобы анимация заработала, нужно обязательно указать два подсвойства:

**animation-name** — имя набора кейфреймов;

**animation-duration** — хронометраж анимации; время, за которое анимация проигрывается целиком.

div {

animation-name: move;

animation-duration: 2s;

}

Такой код запустит анимацию из кейфреймов с именем **move** и завершит её через две секунды. Можно также использовать сокращенную запись:

div {

animation: move 2s;

}

@keyframes move\_bg{

from{

background-position: 0 0;

}

to {

background-position: 100px 0;

}

}

.demo\_\_row\_size\_l {

background-image: url('https://pictures.s3.yandex.net/animation\_topic/line\_l.svg');

animation: move\_bg 8s linear infinite alternate;

}

В группе свойств animation есть много подсвойств, которые производят тонкую настройку анимации:

* **animation-timing-function** — функция времени. Аналогичное мы рассматривали для свойства transition. Можно задать в виде куба Безье или ключевыми словами (**linear, ease-in, ease-in-out, ease, ease-out**).
* **animation-delay** — задержка перед стартом анимации, в секундах.
* **animation-iteration-count** — сколько раз запустить анимацию (англ. iteration-count, «счёт итерациям»). Задаётся целым числом, но существует специальное ключевое слово **infinite**, чтобы воспроизводить анимацию бесконечно.
* **animation-direction** — направление анимации. Значений четыре, задаются ключевыми словами. Можно запустить от начала к концу — значение **normal**, с конца до начала — **reverse**, от начала к концу и обратно — **alternate**, от конца к началу и обратно — **alternate-reverse**.
* **animation-fill-mode** — указывает, как элемент ведёт себя после завершения анимации. Значение **forwards** сохранит элемент в том состоянии, в котором он оказался после выполнения анимации; **backwards** — вернёт состояние до начала анимации; **both** (то и другое) — самый сложный вариант. Он сначала установит элементу состояние первого кадра, которое сохраняется до старта анимации — столько времени, сколько задано в **animation-delay**. По окончании анимации устанавливается состояние последнего кадра.

.demo\_\_item-ico {

animation: pulse 1s ease-in-out infinite alternate;

}

## **Кроссбраузерность**

**Кроссбраузерность** - это одинаковая работа и отображение сайта во всех браузерах.

**Вендорный префикс** — это специальный указатель для браузера, что мы пишем свойство именно для него. Вендорных префиксов не так много:

**-moz-свойство** — префикс для Firefox;

**-webkit-свойство** — префикс для Safari и Chrome;

**-ms-свойство** — префикс для Internet Explorer и Edge;

**-o-свойство** — для поддержки браузерами Opera.

Некроссбраузерные свойства принято писать с нужными префиксами, до объявления основного свойства:

.my-class {

-webkit-box-shadow: /\* какие-то свойства тени \*/;

box-shadow: /\* какие-то свойства тени \*/;

}

Если браузер не знает свойство с префиксом, он проигнорирует его и перейдёт к следующей записи.

Для **flex** это выглядит так:

.my-class {

display: -webkit-box;

display: -ms-flexbox;

display: flex;

}

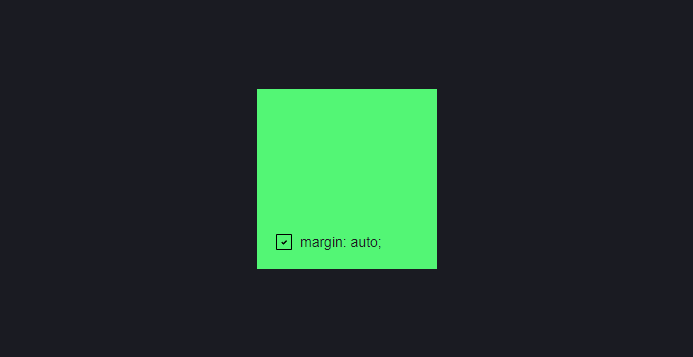
# **display:flex**

Свойство **display: flex;** превращает элемент во флекс-контейнер (*англ. flex container*), а вложенные в него дочерние — во флекс-элементы (*англ. flex items*).

Флекс-контейнер управляет не элементом, а пустотой вокруг него.

Если свойство содержит уточнение — например, margin-left: auto; — вся пустота оказывается слева, а элемент прижимается к правому краю контейнера.

А margin: auto; оказывает совершенно магическое действие. Пустота равномерно окружает элемент, и он оказывается точно посреди своего контейнера как по ширине, так и по высоте.

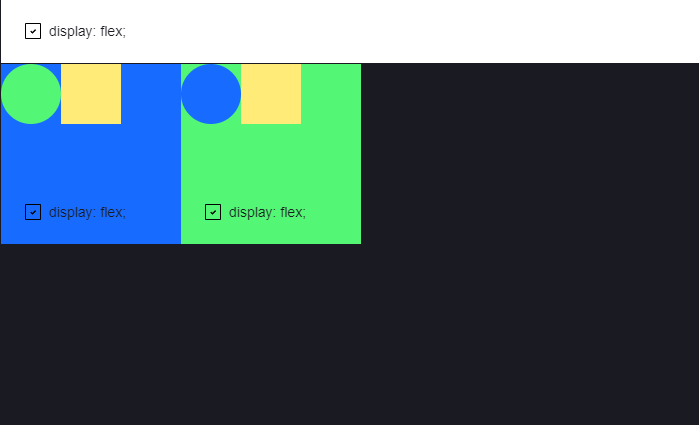


**флекс-контейнер** - родительский элемент: гибкое пространство, в котором элементы распределяются по правилам, установленным для контейнера;

**флекс-элементы** - дочерние элементы флекс-контейнера; подчиняются правилам, описанным как для контейнера, так и для отдельных элементов персонально.

## **Вложенные флекс-контейнеры**

Вложенный во флекс-контейнер элемент может сам стать флекс-контейнером. Так он сможет распоряжаться вложенными в него элементами. Заметим, что флекс-элементы подчиняются только своему «непосредственному начальнику» (родителю) — настройки флекс-контейнеров более высокого уровня до них не доходят.



## **Направление внутри флекс-контейнера**

По умолчанию флекс-элементы внутри контейнера отображаются в ряд слева направо в том порядке, в каком они прописаны в коде HTML-документа.

**flex-direction: row;** — обычный ряд (англ. row, «ряд»), элементы следуют в потоке, слева направо. Это значение по умолчанию, его можно и не прописывать.

**flex-direction: row-reverse;** — ряд в обратном порядке (англ. reverse, «обратный ход»).

**flex-direction: column;**— колонка (англ. column, «столбец») в обычном порядке. Очень пригождается при адаптации под мобильные устройства.

**flex-direction: column-reverse;** — колонка в обратном порядке.

## **Управление положением содержимого флекс-контейнера**

Регулировка вдоль строки либо колонки задаётся свойством **justify-content**.

**justify-content: flex-start:** сначала идут флекс-элементы, за ними — оставшееся пустое пространство. Если значение **flex-direction — row**, все флекс-элементы прижмутся к левому краю контейнера. А справа окажется пустота. Если **flex-direction: column**, элементы идут сверху вниз. И **свойство justify-content: flex-start** соберёт флекс-элементы сверху, а пустое пространство окажется снизу. **justify-content: flex-start** — это значение по умолчанию.

**justify-content: flex-end:** сначала идёт пустое пространство, а к концу контейнера прижаты элементы. При flex-direction: row, элементы окажутся справа, а всё пустое пространство — слева. При flex-direction: row-reverse — наоборот.

**justify-content: center:** флекс-элементы собраны посередине, а пустое пространство распределено поровну с двух сторон от них.

**justify-content: space-between:** первый элемент прижат к началу контейнера, последний — к концу, а остальные расставлены между ними с одинаковыми промежутками. Подходит для меню, растянутого во всю ширину страницы.

**justify-content: space-around:** всё пустое пространство делится на равные промежутки. Каждый флекс-элемент получает отступы по обе стороны, равные этому промежутку. Так между флекс-элементами оказывается отступ в 2 промежутка, а перед первым и после последнего элементов — в 1 промежуток.

**justify-content: space-evenly:** все отступы равной величины. И между элементами, и между границами контейнера и первым и последним элементами.

## **Выравнивание флекс-элементов в контейнере**

Когда флекс-элементы разной высоты выстроены в ряд, или столбец составлен из элементов разной ширины, их также можно выровнять по определённой линии. Предназначенное для этого свойство так и называется **align-items**, в переводе с английского «выровнять элементы по линии».

**align-items: stretch** («растянуть») — установлено по умолчанию. Вместо выравнивания элементы растянуты от одного края контейнера до другого: в ряду сверху донизу, в колонке справа налево. Значение stretch стало большим достижением флексбокс-вёрстки. Оно позволяет придать всем элементам одинаковый размер, не прибегая к определению ширины и высоты.

**align-items: flex-start** (по началу флекс-контейнера) — когда направление контейнера row или row-reverse, все элементы прижаты к верхнему краю контейнера, а при значениях column или column-reverse — к левому.

**align-items: flex-end** (по концу флекс-контейнера) — когда направление контейнера row или row-reverse, все элементы прижаты к нижнему краю контейнера, а при значениях column или column-reverse — к правому.

**align-items: center** («центрировать») — центральная ось контейнера делит каждый элемент пополам.

**align-items: baseline** («по базовой линии») — базовые линии всех флекс-элементов становятся продолжением базовой линии первого по порядку.

## **Перенос элементов при сжатии окна браузера**

По умолчанию флекс-контейнер делает окно "резиновым": при сжатии или расширении окна просмотра браузера отдельные флекс-элементы автоматически настраивают свою ширину или высоту так, чтобы поместиться в контейнер. Это поведение можно изменить так, чтобы при сжатии элементы переносились на новую строку или колонку. Для этого свойству flex-wrap задают значение:

**wrap** (англ. wrap, «завернуться»). У флекс-контейнера с таким свойством последние элементы заворачиваются (переходят) на новую строчку, если не умещаются на текущей.

**wrap-reverse** («перенос наоборот») позволяет устроить так, чтобы переносились не последние, а первые элементы строки/колонки.

**flex-wrap** есть у флекс-контейнера всегда, только по умолчанию ему задано значение **nowrap**, запрещающее перенос.

## **Свойства флекс-элементов. Порядок**

Изменение порядка отображения элементов без редактирования HTML коду осуществляется через добавление свойства **order.** Элемент, которому задано CSS-свойство **order: 1;** будет показан раньше элемента со свойством **order: 2;** (здесь могут быть любые целые числа, лишь бы одно из них было меньше другого).

Элемент с отрицательным значением порядка (**order: -1;**) будет первым в контейнере, если у других элементов значения положительны или даже не заданы И наоборот, если порядок с положительным значением задан одному-единственному элементу, тот закрепляется на последнем месте.

## **Свойства флекс-элементов. Выравнивание отдельного элемента**

**align-self** испльзуется чтобы отдельный элемент не подчинялся общему правилу, оно применяется непосредственно для уникального флекс-элемента и принимает те же значения, что **align-self**:

**align-self: stretch** («растянуть») — установлено по умолчанию. Вместо выравнивания элементы растянуты от одного края контейнера до другого: в ряду сверху донизу, в колонке справа налево. Значение stretch стало большим достижением флексбокс-вёрстки. Оно позволяет придать всем элементам одинаковый размер, не прибегая к определению ширины и высоты.

**align-self: flex-start** (по началу флекс-контейнера) — когда направление контейнера row или row-reverse, все элементы прижаты к верхнему краю контейнера, а при значениях column или column-reverse — к левому.

**align-self: flex-end** (по концу флекс-контейнера) — когда направление контейнера row или row-reverse, все элементы прижаты к нижнему краю контейнера, а при значениях column или column-reverse — к правому.

**align-self: center** («центрировать») — центральная ось контейнера делит каждый элемент пополам.

**align-self: baseline** («по базовой линии») — базовые линии всех флекс-элементов становятся продолжением базовой линии первого по порядку.

## **Свойства флекс-элементов. Базовая ширина элемента**

Иногда необходимо, чтобы один флекс-элемент имел определённый размер, пока другие свойства флекс-контейнера не начнут его сжимать или расширять. Для этого есть **свойство flex-basis** (*от англ. «базис», базовый размер флекс-элемента*). Значение задаётся в пикселях.

Это ширина, если флекс-элементы отображаются в ряд, и высота — если в колонку.

# **display: grid**

## **Grid**

**Грид** — это макет для сайта (его схема, проект).

Грид-модель позволяет размещать контент сайта (располагать его определенным образом, позиционировать). Она позволяет создавать структуры, необходимые для обеспечения отзывчивости сайтов на различных устройствах.

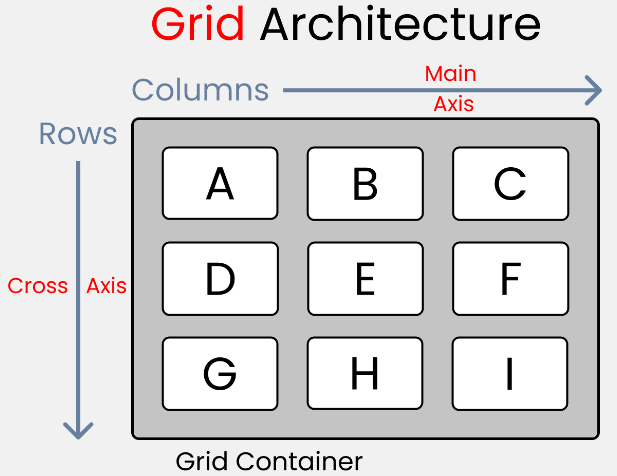
**Грид-контейнер** (англ. grid container) — набор вертикальных и горизонтальных линий, которые пересекаются и создают области, в которых могут располагаться грид-элементы — все прямые потомки грид-контейнера. Кроме значения **display: grid** у него есть ещё одно — **inline-grid**. В этом случае грид-контейнер будет вести себя как строчный элемент и занимать только то пространство, которое необходимо, чтобы разместить грид-элементы.

**Грид-элементы** (англ. grid items) — элементы, которые располагаются в грид-контейнере. То есть прямые потомки того элемента, которому мы задаём **display: grid;**.

**Грид-линии** (англ. grid lines) — невидимые вертикальные и горизонтальные линии внутри грид-контейнера, которые пересекаются и создают строки и колонки. Линии имеют нумерацию начиная с единицы.

**Грид-колонки** и **грид-строки** (англ. grid tracks) — результат пересечения грид-линий, то есть пространство, заключённое между ними. Грид-колонки и грид-строки задают сетку страницы.

**Грид-ячейки** (англ. grid cells) — получаются в результате пересечения строк и колонок.

Элементы Грида (grid items) располагаются вдоль главной или основной (main) и поперечной (cross) оси (axis). При помощи различных свойств мы можем манипулировать элементами для создания макетов. 

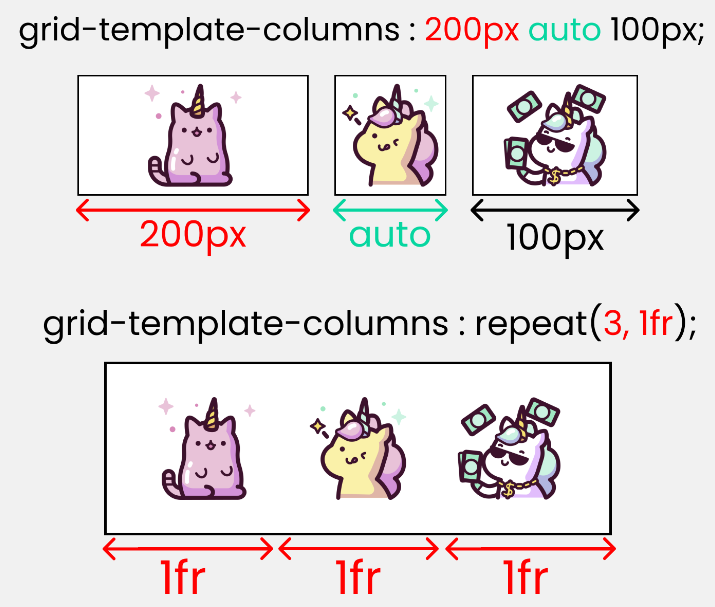
Помимо прочего, у нас имеется возможность объединять строки и колонки.

**Схема CSS Grid** содержит все возможные свойства, предоставляемые Гридом. Эти свойства делятся на:

* родительские (свойства грид-контейнера)
* дочерние (свойства грид-элементов)

## **Родительские свойства grid**

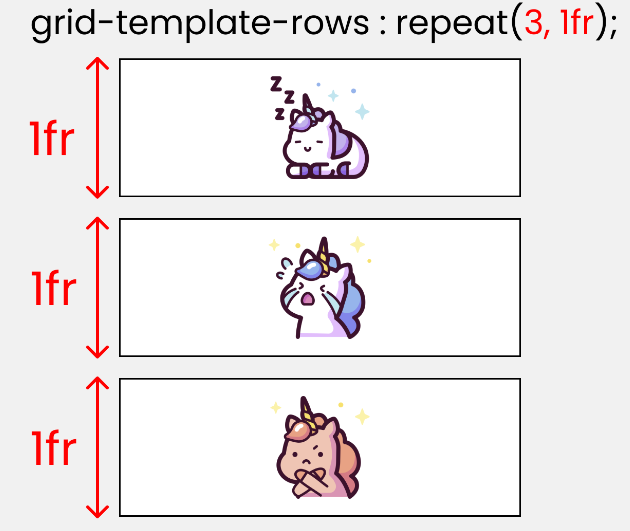
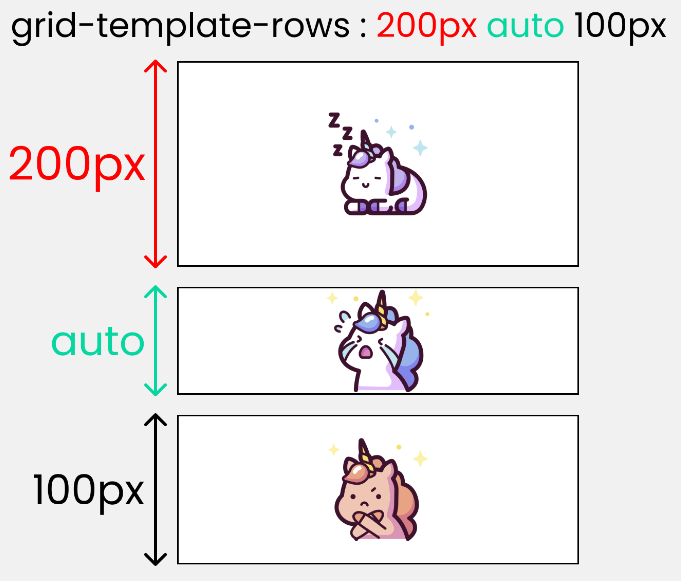
**grid-template-columns** - используется для определения **количества** и **ширины** колонок. При этом, можно определять как свойства для каждой колонки в отдельности, так и устанавливать ширину всех колонок с помощью функции **repeat().**



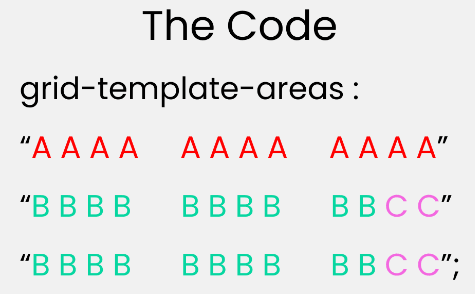
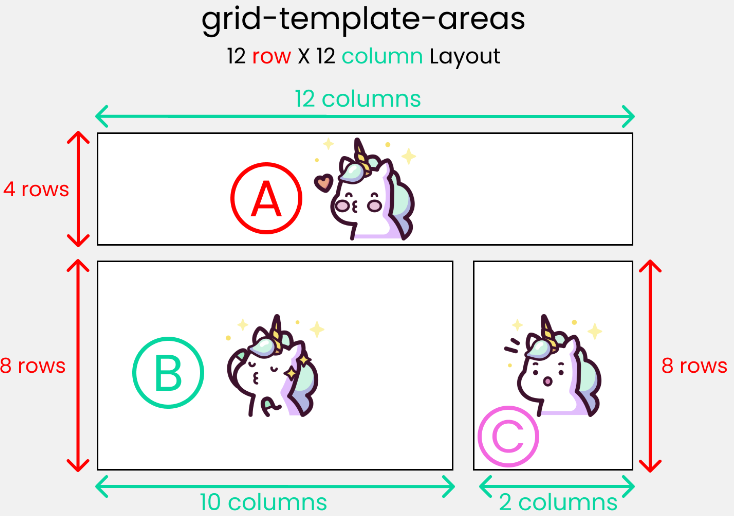
*значения в пикселях будут точными. Ключевое слово auto означает заполнение элементом всего доступного пространства.*

*использование единицы fr (фракция) в repeat() означает, что все контейнеры будут иметь одинаковую ширину.*

**grid-template-rows -** используется для определения **количества** и **высоты** строк. При этом, можно определять как высоту каждой колонки в отдельности, так и устанавливать высоту всех строк с помощью функции **repeat().**



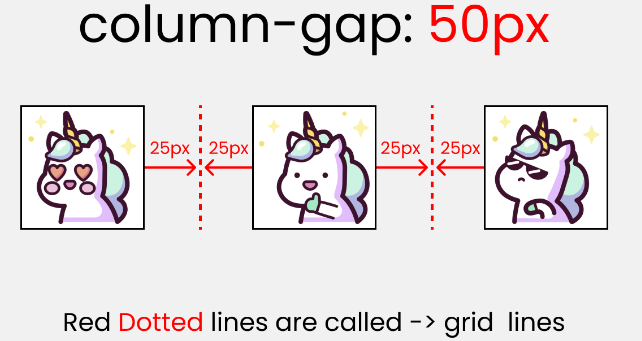
**grid-template-areas -** используется для определения количества пространства, занимаемого ячейкой Грида (grid cell), в терминах колонок и строк, в родительском контейнере.



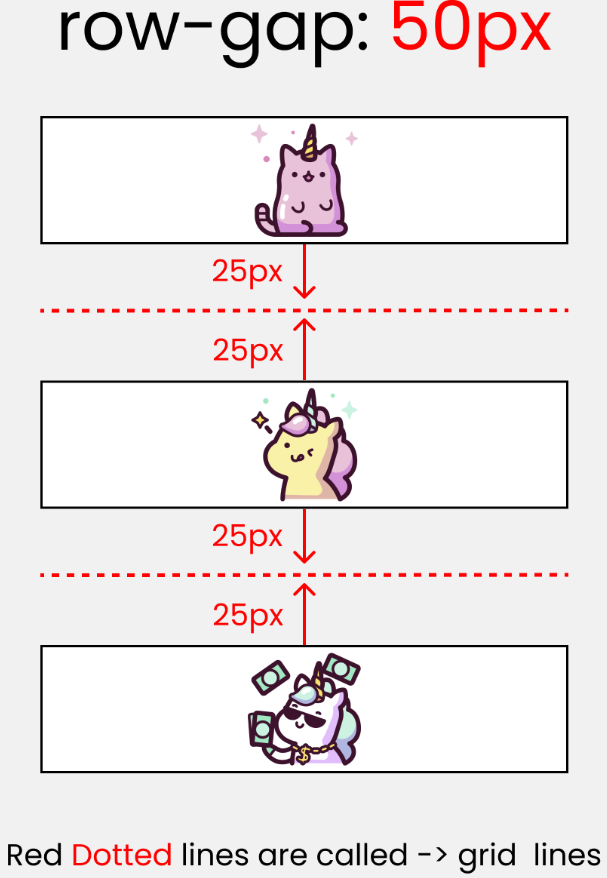
Для получения результата требуется не только родительское, но и хотя бы одно дочернее свойство:

* grid-template-areas: родительское свойство, создающее схему
* grid-area: дочернее свойство, которое использует схему

**column-gap -** Данное свойство используется для добавления отступа между **колонками**.

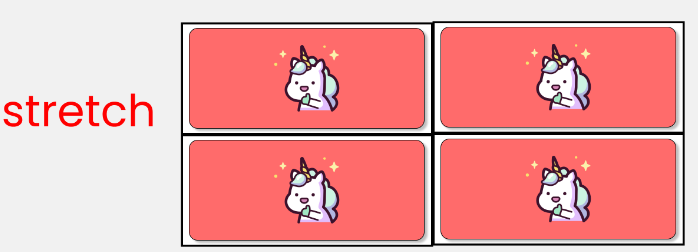


**row-gap** - используется для добавления отступов между **строками**.



**justify-items** - используется для позиционирования грид-элементов внутри грид-контейнера вдоль **главной оси**. Оно принимает **4** возможных значения:

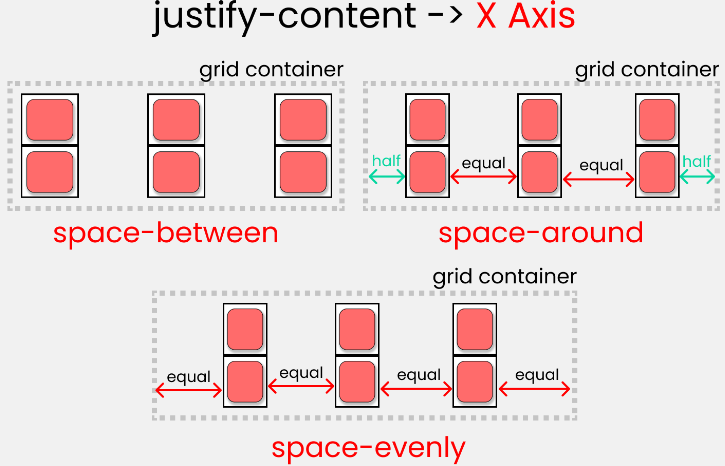
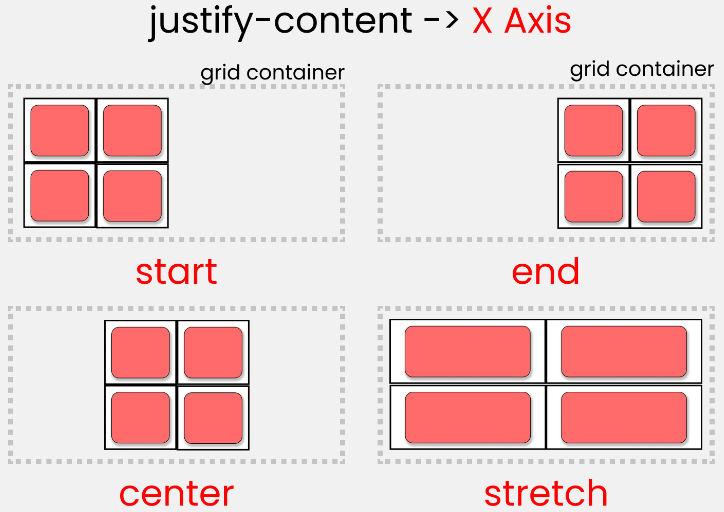




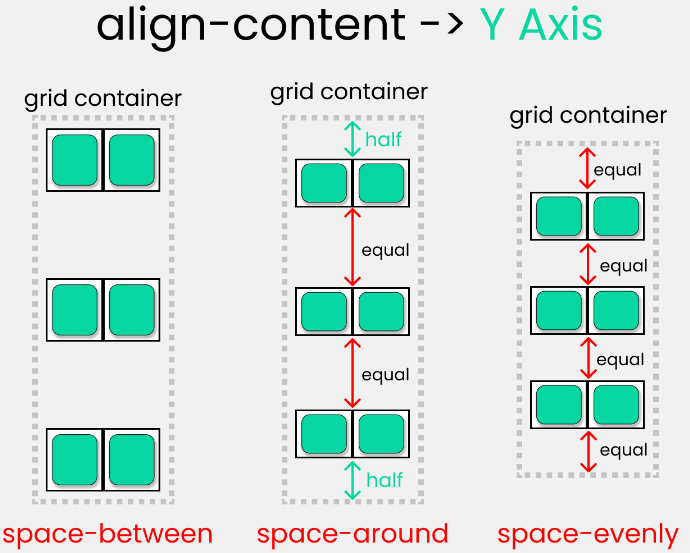
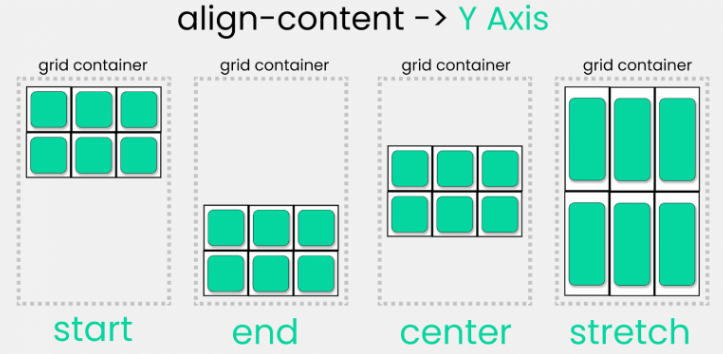
**align-items** - используется для позиционирования грид-элементов внутри грид-контейера вдоль **поперечной оси**. Оно принимает **4** возможных значения:



**justify-content** - используется для позиционирования самого грида внутри грид-контейнера вдоль **основной оси**. Оно принимает **7** возможных значений:



**align-content** - используется для позиционирования самого грида внутри грид-контейнера вдоль **поперечной оси**. Оно принимает **7** возможных значений:



# **Адаптивная верстка**

Есть два подхода к созданию интерфейса, подстраивающегося под разную ширину устройств: «**резиновая**» вёрстка и **адаптивная** вёрстка.

**При «резиновой» вёрстке** элементы растягиваются: пропорционально масштабируются под размеры окна, но не меняют своё положение и внешний вид.

**Адаптивная вёрстка** предполагает изменение внешнего вида элемента в зависимости от размера браузера.

Секрет «резиновых» макетов в том, что размеры элементов задаются в относительных единицах измерения. В адаптивных макетах внешний вид элементов меняется по специальным правилам, прописанным для разных состояний браузера.

Существует термин «**отзывчивый дизайн**», который часто противопоставляют «**адаптивному дизайну**». При любом из подходов сайт верстают в нескольких состояниях: для телефона, планшета, ноутбука, десктопа. Но если менять размер окна браузера, отзывчивый сайт переходит из одного состояния в другое плавно, а при адаптивном — скачками.

Без применения JavaScript есть только три способа отреагировать на изменение размера окна браузера:

* применяя относительные единицы измерения, сделать элемент резиновым, задав его размеры в относительных единицах: %, vh, vw;
* писать CSS-правила, срабатывающие при разных условиях: одна ширина элемента при одном размере окна, другая — при другом;
* показать или скрыть элемент при определённых условиях. Это частный случай применения второго способа, например значение свойства display меняют на none и наоборот).

Чтобы добиться хорошего результата вёрстки, нужно следовать трём правилам:

* создавать «резиновую сетку»;
* создавать «резиновые блоки»;
* изменять CSS-стили в контрольных точках.

Анализируя макет, обращайте внимание:

1. Как меняется количество строк и колонок при сжатии элементов?
2. Меняют ли элементы свой порядок относительно друг друга?
3. Как меняются размеры и отступы? Попробуйте найти или предположить закономерности.
4. Какие элементы пропадают и появляются на разных разрешениях?
5. Как изменение элементов связано с изменением стилей их контейнеров? Например, **свойство flex-direction** часто меняют с **row** на **column**, а спозиционированные элементы возвращаются в поток.

# **JS**

## **Ништяки**

<*script src="путь к файлу"></script> -* подключение js к html перед закрывающим тегом </body>.

*<script* ***async*** *src=’script.js’ type=’text/javascript’></script> -* загружает скрипт фоном, актульно при подключении скрипта в head

*<script* ***defer*** *src=’script.js’ type=’text/javascript’></script> -* загружает скрипт фоном, но сохраняет порядок загрузки и порядок выполнения скриптов

## **Переменные**

Переменную объявляют через **var** or **let.** Регистр в имени переменной важен.

Типы данных:

* Number **var number = 42**
* String **var string = ‘Message’**
* Boolean **var isTrue = true // false**
* Object **var obj = {a : 1}**
* Null **var nothing = null**
* Undefined **undef = undefined**

**Для чисел доступны операции:**

**console.log(2 + 2);** *// выведет в консоль 4*

**console.log(2 \* 2);** *// выведет в консоль 4*

**console.log(2 - 2);** *// выведет в консоль 0*

**console.log(2 / 2);** *// выведет в консоль 1*

**console.log(2 \*\* 3);** *// возведение в степень. выведет в консоль 8*

**Typeof** возвращает тип переменой. Используется через пробел **(typeof number)** или в скобках **typeof(string)**, *принято писать через пробел.*

***Типы данных str & int можно конкатинировать: ‘hello’ + 12 => hello12. Так же с Boolean, если булиан со стракой, то преобразовывет все в строку, а при числех складывает true = 1, false = 0.***

## **Массив**

Массив объявляется: **let m = [‘bla’, ‘bubu’, ‘tutu’] or var array = [‘1’, ‘2’,'3',’4’]**

**m.push(‘eto’)** – добавляет элемент в конец массива

**m.pop()** – удаляет и возвращает последний элемент массива

**m.shift()** – удаляет и возвращает первый элемент

**m.unshift()** – добавляет элемент в начало массива

**m.indexOf(‘bla’)** – поиск индекса элемента, если элемента нет, возвращает -1

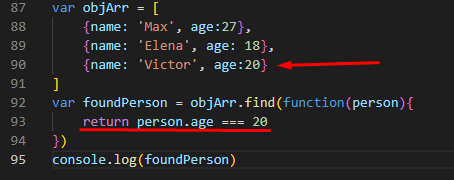
**array.join(‘;’)** – соединитель для перевода массива в строку, в скобках принимает символ для соединитель

**array.reverse()** – изменить порядок вывода символов

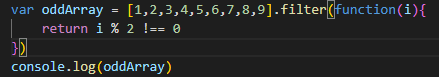
**array.splice(стартовый индекс, сколько элементов удалить, что вставить вместо удаленных)** – выводит массив с указаного индекса с учетом удаленного количества элементов и замены. **array.splice(0, 3, ‘yo’).** Передавать на вставку элементов можно по необходимости **array.splice(0, 0, ‘yo’, ‘q’, ‘w’)**.

**m.concat(пустое значение или другой массив)** – клонирует массив, можно использовать что бы соеденить два массива

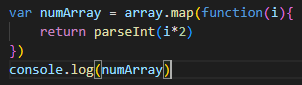
**.find** – возвращает искомый объект, может принимать функцию



**.filter()** – фильт значений, может принимать функцию



**.map()** – возвращает новый массив, принимает функцию



## **Операторы**

**%** - остаток от деления

**i++** or **i--** инкримент и дикремент

**++i** or **--i** –сначала выводит прошлое значение потом инкремент и дикремент

**4 == 5** – оператор сравнения только на равенство значений, включая строку 4 == ‘4’

**4 != 5** – оператор равнения на не равенство

**4 === ‘4’** – оператор сравнения значения и типа данных

**&&** - и, вернет true если оба варианта true (true && true)

**||** - или, вернет true если хотя бы один вариант true (true || false)

**!** – не, инвертирует true или false (!true вернет false)

## **Условные операторы**

Инструкция **if (…)** вычисляет условие в скобках и, если результат true, то выполняет блок кода, если хотим выполнить более одной инструкции, то блок кода заключается в **{…;}. Рекомендуется использовать фигурные скобки всегда.**

Можно добавлять дополнительные состояния.

**if(..){..}**

**else if (..){..;}**

**else {..;}**

**0, null, undefined, пустая строка(‘’), NaN в условии if возвращает false.**

Так же можно не использовать else. Как пример нужен вывод всех четных чисел из массива:

**var numbers = [1, 2, 3, 4 , 5, 6]**

**for (ver i = 0; I < numbers.length; i++) {**

**if (numbers[i] % 2 !== 0) {**

**continue**

**}**

**console.log(numbers[i])**

**}**

## **Тернарные операторы**

**Тернарный оператор –** единственный оператор в js, принимающий три операнда: условие, за которым следует ?, затем выражение, которое выполняется если условие истина, сопровождается двоеточием и, наконец выражение, котторое выполняется если условие ложно

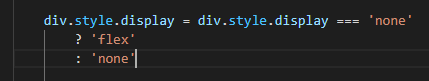
**let result = Условие ? значение1 : значение2;**

*a > b*

*? console.log(‘первое число больше’)*

*: console.log(‘первое число не больше’*)

*вложение if в переменную*



*Если дисплей равен нон, то добавляем флекс, если нет то ставим нон*

## **Несколько операторов «?»**

Последовательность операторов вопросительного знака ? позволяет вернуть значение, которое зависит от более чем одного условия.

**let age = prompt(‘old?’, ‘18’);**

**let msg = (age < 3) ? “Hello kid” :**

**(age < 18) ? ‘Hi!’ :**

**(age < 100) ? ‘Hello old man’:**

**‘omfg why are u live?’**

**alert(msg);**

Смысл оператора «вопросительный знак» ? – вернуть то или иное значение, в зависимости от условия. Пожалуйста, используйте его именно для этого. Когда вам нужно выполнить разные ветви кода – используйте if.

## **Оператор объединения с null ??**

Оператор **??** возвращает первый аргумент, если он не **null/undefined**, иначе второй. Оператор объединения с **null** не является чем-то принципиально новым. Это всего лишь удобный синтаксис, как из двух значений получить одно «определённое».

Как правило, оператор **??** нужен для того, чтобы задать значение по умолчанию для потенциально неопределённой переменной. Пример:

**let user;**

**alert(user ?? "Аноним");** *// вернет Аноним, т.к. для перемоной* ***user*** *нет значения*

*Если же в переменную добавить значение, то вернется значение.*

**let user = "Иван";**

**alert(user ?? "Аноним");** *// Иван*

## **Switch Case**

**Switch (переменная) {**

**case ‘значение для сравнения с переменной’:**

**дейсвие если совпадает**

***break\****

**}**

**case ‘значение для сравнения с переменной’:**

**Действие если совпадает**

***break\****

**default:**

**Дейсвие если ни одно условие не совпало**

\*Что бы при совпадении одного условия, выйти нужно в конце блоков прописать **break**

## **Функции**

**Функция** — это блок кода, который можно переиспользовать много раз. Создают функцию ключевым словом **function**

Функция принимает параметр, параметры и что то возвращает. Функцию можно добавлять в if ***( if (название функции < 10){}…).***

*Функция создается двумя способами(При создание функции через переменную, сначала надо создать функцию, только потом обращаться):*

**function название функции (входящий параметр) {**

**инструкции, могут быть переменные**

**return переменная или значение которое возвращает функция**

**}**

*function old(year) {*

*var currentYear = 2022*

*var result = currentYear – year*

*return result*

*}*

**Объявление переменной = function (входящий параметр) {**

**Инструкции, могут быть переменные**

**return**

**}**

*var I = function (name) {*

*console.log(‘Hi,’ + name)*

*}*

## **Строки**

Принято хранить строку в одинарных ковычках.

Экранировать, обратным слешь «\».

**Методы строк:**

var str = ‘Hello world’ or var str = ‘1, 2, 3, 4, 5, 6’

**str.length** – длина строки

**str.toUpperCase()** – Все буквы заглавные.

**str.toLowerCase()** – Все буквы строчные.

**str.charAt(1)** – Вызов элемента по индексу (e)

**str.indexOf(‘world’)** – Возвращае индекс по элементу (6), вернет -1 если элемента нет.

**str.substr(1, 4)** – Возвращает срез строки, начальный индекс и количество символов.

**str.substring(1)** – Возвращает срез строки, начальный и конечный индекс не включительно(не обезательно)

**str.split(‘,’)** – разделяет строку согласно переданому разделителю, в примере это запятая

**str.trim()** – удаляет лишние пробелмы в начале и конце

## **Числа**

var delta = 2.666

**delta.toString()** – переводит число в строку.

**delta.toFixed(1)** – округляет в большую сторону, до указаного количества знаков после запятой .

**Что бы строку перевести в число, нужно перед переменной ставить +.**

**parseFloat(delta. toFixed(1)) –** вернет первое число в строке

**parseInt(delta) –** вернет первое целое число в строке

**isNaN(число, переменная) –** проверка является ли число валидным, если передано число вернет false

## **Объект Math**

**Math.random** – генерирует рандомное число, до 1 не включетельно. Что бы получить целое число необходимо **Math.random() \* [длина массива]** *(т.е. если нужно от 0 до 5, то умножаем на 6 и тд.)*

**Math.floor** – округляет в меньшую сторону

**Math.ceil** – окгруляет в большую сторону

Пример:

Необходимо получать целые рандомные числа от о до 10:

let randomNumberInt = Math.floor(Math.random() \* 11);

## **Объекты**

**Объекты** - хранения разнородных данных, создаются фигурными скобками. Объект состоит из ключ: значение. *Объект — ящик с элементами. Фигурные скобки — стенки ящика, границы объекта. Лежащие внутри ящика элементы разделены запятыми.*

**var person = {**

**name = ‘Василий’,**

**year = 1990,**

**family: [‘Елена’, ‘ Игорь’],**

**car: {**

**year: 2010,**

**model: ‘Ford’**

**}**

**calculateAge: function() {**

**var age = 2022 – this.year** (person.year, this используется если вызывается переменная в из общего объекта) или **this.age = 2018 – this.year** (добавляем в значение age данные)

**console.log(‘Возраст’, age)**

**}**

**}**

Обратиться к свойству объекта можно:

**person.name** => ‘Василий’

**person[‘year’]** => 1990

**person[‘car’].year** => 2010 или **var field = ‘car’ person[filed].year** => 2010

Замена данных:

**person.year** = 1993 => изменит данные у ключа year

## **JSON**

**JSON.stringify()** – переводит объект в строку формата json

**JSON.parse()** – переводит json в объект

## **Дата и время**

**new Date** – базовый метод для работы с датой и временем, фиксирует время на момент создания

## **Циклы**

Цикл позволяет повторять похожие действия, задав условия повторения. Конструкция цикла похожа на конструкцию условия **if**. Сначала идёт ключевое слово **for**, потом круглые скобки, а затем фигурные. В фигурных скобках размещают код, который нужно повторять:

**var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]**

***for (счетчик, условие выполнения, итерация) {***

***действие***

***}***

***for (var i = 0; I < numbers.length; i++){***

***if (numbers[i] % 2 === 0) {***

***console.log(numbers[i])***

***}***

***}***

## **Прототипы**

Функции, которые являются классом принято писать с заглавной.

**function Car(name, year) {**

**this.name = name**

**this.year = year**

**}**

**var ford = new Car(‘Ford’, 2015)**

*Благодаря прототипам можно создать метод.*

*Для примера создаем метод который возвращает возраст машины:*

**Car.prototype.nameOfMethod = function(){**

**return new Date().getFullYear() – this.year**

**}**

*Что бы вызвать новый метод достаточно написать:*

**ford. nameOfMethod() => 7**

## **DOM**

**DOM** – Document Object Model, структура построения страницы.

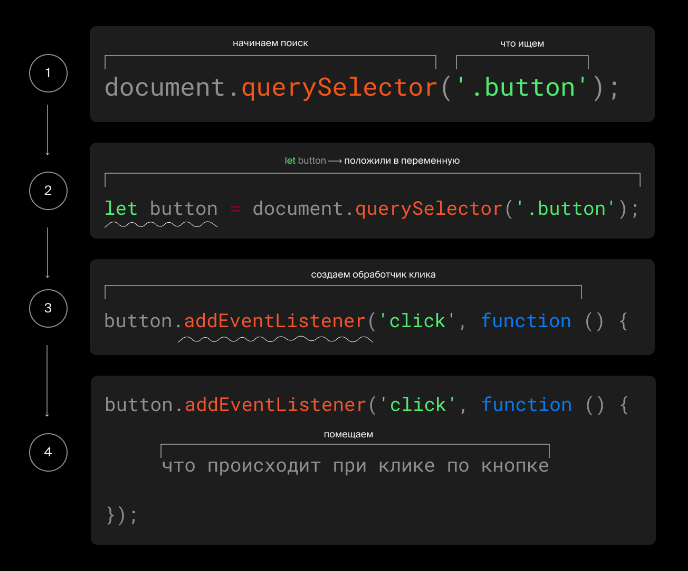
**document** – Что бы работать с элементами страницы, в нем вся информация о веб-странице: URL-адрес, таблицы стилей, кодировка, гиперссылки, все тексты. Через него можно получить доступ к управлению любыми элементами на странице.

## **Доступ к элементам HTML**

**.querySelector()** – инструмент, работает как функция и принимает на вход селектор класса с которым хотим взаимодействовать

**элемент.addEventListener('событие', function () { что-то происходит });** **–** Слушатель событий у элемента ,когда с элементом совершается событие запускается функция.

Пример с кнопкой:



*Устаревший способ:*

**document.getElementById(id тега) –** получить доступ к элементу по тегу. Если задать как значение переменной то его можно вывести в консоль

**var i = document.getElementById(‘id\_name’)**

**console.log(i)**

**document.getElementsByClassName(‘name\_of\_class’) -–** получить доступ к массиву элементов по имени класса

**document.getElementsByTagName(‘h1 or p etc.’)** **-–** получить доступ к массиву элементов по имени тега

*Принято получать доступ к элементам DOM:*

query selector работает по принциму css т.е. объявляются классы, теги и тд как css, тоже самое и для дочерних, например для получения доступа к списку, который находится в в классе .classname (.classname ul)

**var el = document.querySelector(‘#id .classname h1’) – получить доступ к одному элементов**

**ver el = document.querySelectorAll(‘.classname’) – получить доступ к массиву элементов**

**el.innerHTML -** можно получить содержимое элемента, через него можно изменить содержание пример: **el.innerHTML = ‘<h2> Hello world</h2>**

**el.textContent** – возвращает текст элемента и изменяет содержание, пример:

**el.textContent = ‘This is base’**

**переменная.removeAttribute(‘имя аттрибута’) –** удалить аттрибут

**document.createElement(‘div’)** – создание элемента

**переменная.insertAdjacentElement(‘позиция’, элемент)** – порядок отображения элементов при добавлении 

## **Атрибуты**

*<a id=”link1” href=”google.ru” target=”\_blank” title=”Click me!”>google</a>*

**var a = document.querySelector(‘a’)**

**a.getAttribute(‘href’)** – получить значение по атрибуту

**a.setAttribute(‘href’, ‘ya.ru’) или (‘title’, ‘Go to yandex.ru’) –** Изменить значение аттрибута, сначала передает аттрибут который изменяем, а потом новые данные

## **Классы**

*var a = document.querySelector(‘#link1’)*

**a.classList.add(‘new\_class’)** – добавление класса

**a.classList.remove(‘old\_class’)** – удаление класса

**a.classList.contains(‘name of class’)** – проверка есть ли класс, возвращает true or false

## **Добавление событий**

*Два метода добавления функции к событию:*

*Первый*

**button.addEventListener(‘click’, ссылка на функцию)** – добавление события для кнопки, клик

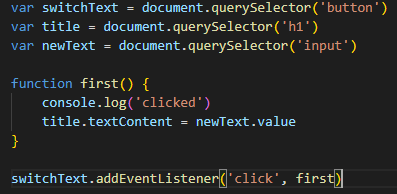
пример:

**function buttonHandler(){**

**console.log(‘clicked’)**

**}**

**button.addEventListener(‘click’, buttonHandler)**



*Второй*

**title.addEventListener(‘mouseenter’, function(){**

**console.log(‘Mouse has entered h1’)**

**this.style.background = ‘red’**

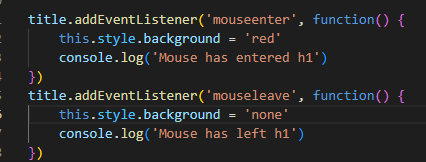
**})**

**button.addEventListener(‘mouseleave’, function(){**

**console.log(‘Mouse has left h1’)**

**this.style.background = ‘none’**

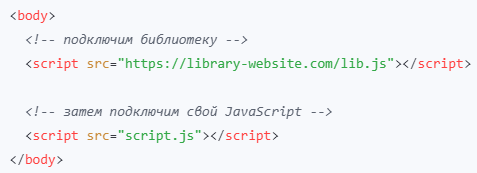
**})**



**В js нельзя писать – в названии переменных, поэтому для использования переменных типо, border-radius, background-color & etc используется кэмелкейс borderRadius, backgroundColor.**

## **Подключение внешних библиотек**

Обычно библиотека — это просто JavaScript-файл, лежащий на стороннем сайте. Потому и подключить её можно как простой файл, указав её адрес в атрибуте **src** тега **script**:



Например библеотека **smoothly.** Эта функция принимает три аргумента — элемент страницы, свойство элемента, которое мы хотим изменить, и новое значение этого свойства.

const element = document.querySelector('.element');

smoothly(element, 'textContent', 'Новое содержимое');

Плавно изменяет содержимое **в примере текст и картинку:**

let phrase = document.querySelector('.phrase');

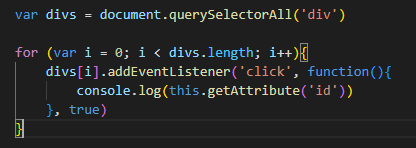
smoothly(phrase, 'textContent', randomElement.text)

let image = document.querySelector('.image');

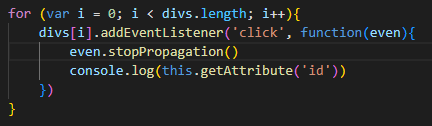
smoothly(image, 'src', randomElement.image)

## **Всплытие и погружение**

js выполняет события на погружение т.е. если создать два дива, друг в друге, один больше чем другой. То при клике на внутрений див, событие будет принимать два действия, сначала клик на внутрений див, а затем на внешний, что бы изменить порядок погружения в метод события **addEventListener** можно передать третье значение true после функции.

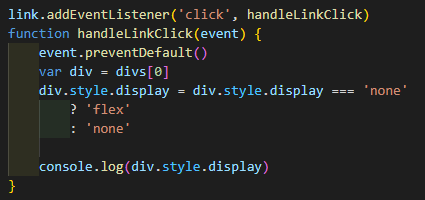


*Дабл клик фиксится:* ***.stopPropagation() останавливает событие.***



***event.preventDefault()*** – реализует функционал переключателя.

Пример к ссылке, добавили событие клик после которого отключается или включается Display: Flex, а переход по ссылке не производится при этом:



## **Делегирование событий**

Вносить изменения в зависимости от события, отследить клик можно через **event.target**



В данном примере, строки с 49 – 52 изменяют цвет у элементов, если клик происходит на элемент p, то цвет меняется на синий, иначе на золотой. с 54 – 56, если у клика есть класс “test”, тогда текст и цвет изменяются.

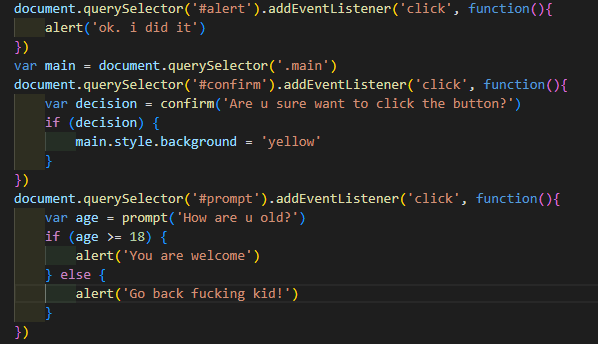
## **Взаимодейсвие с пользователем**

Глобальные методы:

**aler(‘text’)** – окно с уведомление

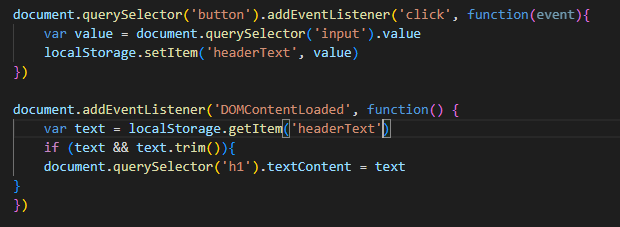
**confirm(‘text’)** – окно с подтверждением или отменой действия( как переменная возвращает true)

**prompt(‘text’, ‘default values’)** – окно с вводом, может содержать значение, сохраняет введенные данные



## **Хранение данных в localStorage**

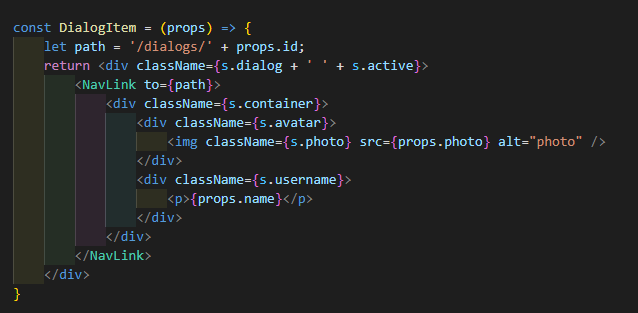
Позволяет сохранять данные в Local Storage.  
Пример для смены h1:



# **React**

## **Старт**

Вся работа с js кодом в реакте выполняется только в {}



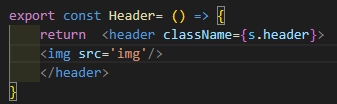
## **Компонента**

**Компонента** – это функция, возвращающая разметку jsx. Так же это тег.

## **Стрелочная функция**

После создания стрелочной функции, в React создается новый тег согласно названию стрелочной функции.

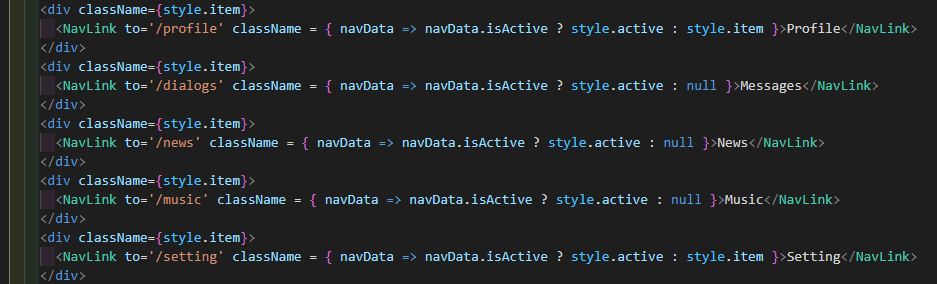
Пример создания стрелочной функции:



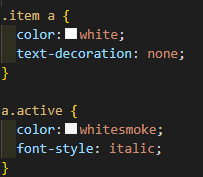
Пример вызова компоненты ранее созданой стелочной функции:

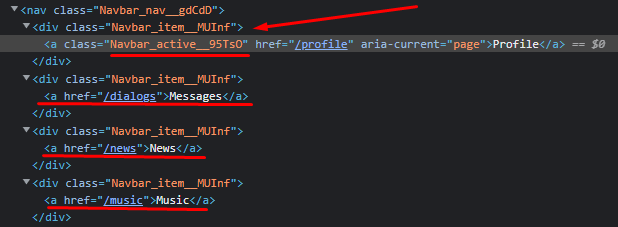


Пример добавления класса active для изменения стиля:



Если ссылка активна(*используется isActive: func - дополнительная логика для определения активности ссылки*), то ей добавляется класс active иначе класса нет (null) при этом стиль прописан в css.





## **Импорта и экспорта**

Что бы импортировать данные нужно:

В родителе прописать **Import { имя } from** ‘./путь’. ./ означает что в текущей папке

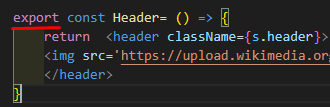
В дочернем прописать **«export»** у функции или «**export default Имя функции**»

Сам элемент вызывается: **<имя />**

Пример импорта компоненты для использоования:

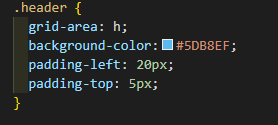
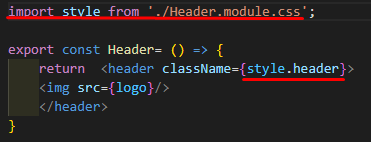


Пример экспорта компоненты для использования:



## **CSS Modules**

**CSS модуль** — это CSS файл, в котором все имена классов и анимаций имеют локальную область видимости по умолчанию. *Ключевая идея здесь — локальная область видимости.*

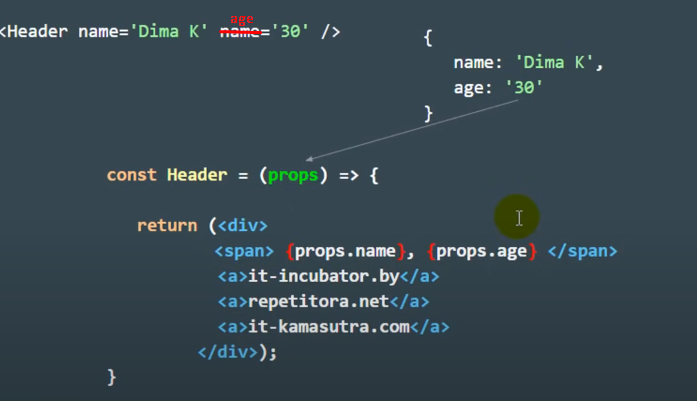


При импорте создаем условно переменную **style**, у которой есть *ключ : значение* -> **header : свойство css**

## **props**

Props представляет коллекцию значений, которые ассоциированы с компонентом. Эти значения позволяют создавать динамические компоненты, которые не зависят от жестко закодированных статических данных.

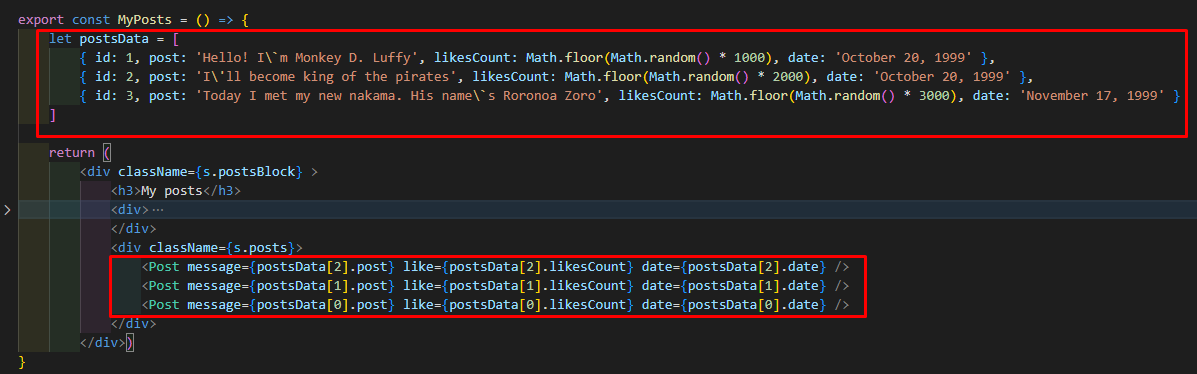
props принимает данные и создает объект с ключ:значение, пример:



Функция принимает props и использует его значения вызовом ключа.



Отправка данных в props:



## **route, browser-router, маршрутизация**

Компонент **Route** просто следит за адресной строкой браузера и если замечает, что там отображается "свой" (нужный ей) URL, то автоматом отрисовывает нужную компоненту.

**npm i react-router-dom -save** – установка пакета маршрутизации

Что бы реакт понимал какой элемент активный:

**Import { BrowserRouter, Routes, Route } from ‘react-router-dom’**

**<BrowserRouter>**

**<Routes>**

**<Route path=’/url’ element= { <Profile /> }**

**</Routes>**

**</BrowserRouter>**

Пример, компонента **App в app.jsx**, возвращает теги страницы, где **Header** & **Navbar** статичны, а в диве **app-wrapper-content** теги отрисовываются в зависимости от **URL**:



**Важно!!** **Route** будет отображать компоненту **Profile** если в **url** есть **profile**, т.е. **url** **profile/spam/blabla** так же вызовет компоненту **Profile.** Это решается добавлением **exact** перед **path -> Route exact path**

## **NavLink**

**NavLink** – улучшенный вариант тега **<a><a/>**, с отличаем что не вызывает перезагрузку страницы, т.е. страница грузится один раз, а дальше подставляются компоненты. ***Работает вместе с Routes.***

**Import { NavLink } from ‘react-router-dom’**

Открытие ссылок без перезагрузки страницы осуществляется:

**<NavLink to =’/url’> Текст </NavLink>**

## **Функция map**

Функция map принимает в себя стрелочкую функцию. Вызывается столько раз, сколько элементов в массиве.

Вызов стилей элементов происходит так же через импорт с созданием имени.

**<div className= {имя импорта.имя класса стиля}>**

# **GIT**

## **Консоль**

**pwd** - Узнать, где вы сейчас

**ls** - содержимое открытой папки

**cd** - Чтобы переместиться из одной папки в другую

**cd "Рабочий стол"** Если в названии папки есть пробелы, используйте кавычки

**..** - Чтобы вернуться в родительскую директорию — то есть на уровень выше

**mkdir** - создаёт папку (**mkdir dev**)

**touch** – Создать файл (**touch index.html**)

**rm** - удалять файлы (**rm about.html**)

**rmdir** - Удалить пустую папку (rmdir images). Если в папке, которую вы пытаетесь удалить, есть какие-то файлы, командная строка не удалит её и выведет сообщение, что папка не пуста.

**rm -r images** – удалить папку и ее содержимое

## **Настройка Git**

**git config –global user.name / user.email** – Задать имя пользователя и почту. (git config --global user.name "Stas Basov")

**git config –list** - список всех свойств конфига

## **Подключаем Git к проекту**

Чтобы «Гит» начал работать в проекте, папку с ним нужно сделать git-репозиторием.

**git init** - инициализация репозитория(в папке)

**git status** - статус файлов в репозитории

**git add** - добавление файла в репозиторий (**git add index.html**)

**git add –all** / **git add -A** – добавление в репозиторий всех файлов.

**git commit -m "мой первый коммит"** – сохранение файлов в репозитории

**git remote add origin url** - синхронизации локального репозитория с удалённым

**git push -u origin main** - заливаем все файлы из локального репозитория в удалённый, который уже привязали.

**git pull** - забрать изменения, сделанные другими разработчиками

**git push** - опубликовать изменения в удалённый репозиторий.

**git log** - история всех изменений репозитория (q выйти)

**git diff** - показывает, какие изменения в коммите: что было сделано и в каких файлах.

**git diff и хеш двух коммитов** - сопоставляет два коммита (git diff a9928ab 11bada1)

**git diff --staged** - посмотреть изменения, добавленные в staged.

**git log –oneline** вывести первые семь символов всех коммитов

**git commit –amend** - добавить файл в уже сделанный коммит

**git reset HEAD -** *Менее радикальный способ откатиться.* Этот подход не затрагивает файлы, с которыми вы работаете. Если файл был изменён или добавлен, но не переведён в staged, сброс изменений этот файл не затронет.

**git reset –hard -** *Более радикальный способ откатиться.*Эта команда удаляет вообще все изменения: и из staged, и из рабочей зоны. После неё вернуть изменения не выйдет. Позволяет вернуться к любому коммиту. Для этого после команды указывают номер коммита - первые 7 цифр его хеша. (git reset --hard bc73f5f)

## **Работать в командной строке быстро**

**Ctrl + A** переходит в начало строки

**Ctrl + E** возвращается в конец строки

**pwd** где мы находи

**cd ~** домашняя директория

**&&** разделение команд (mkdir third-project && cd third-project && touch index.html style.css)

**cp** Копирование (cp что\_копируем куда\_копируем cp index.html /src)

**mv** Перемещение файлов и папок (аналогично **cp**)

**cat** выводит в терминал содержимое текстового файла. (cat index.html)

**cat -n**  выполнить команду cat, пронумеровав строки.

**cat -s**  выполнить команду cat, удалив повторяющиеся строки.

## **Оформление репозитория. README.md**

При публикации проект нужно описать. Так другие разработчики или потенциальные работодатели смогут понять, что представляет собой ваш проект. Такое описание принято указывать в файле README.md.

**Заголовки** разных уровней создают решётками:

Можно добавить черту под заголовком:

Alt-H1

------

**Оформления текста.** Чтобы сделать перенос строки, в конце предыдущей строки должно стоять два пробела.

Чтобы выделить текст **курсивом**, его заключают в астериски (звёздочки) или нижние подчёркивания.

Чтобы сделать начертание **полужирным**, текст окружают двойными астерисками или двойными нижними подчёркиваниями.

Также можно **зачеркнуть** **текст** — для этого его окружают двойными тильдами.

**Ссылки**. Чтобы сделать из текста ссылку, его заключают в квадратные скобки. После текста ссылки в круглых скобках указывают её адрес  
**Что писать в README.md?**

Еесть информация, которую стоит оставлять в README.md почти всегда:

1. Заголовок с названием проекта.
2. Описание проекта: что он делает, какую задачу решает, какую боль закрывает? Если вы выполняете очередную проектную работу, напишите, в чём её отличие от предыдущей.
3. Также в описании можно рассказать о технологиях, которые вы применяли в проекте.
4. Инструкция по использованию проекта и системные требования: версия языка, необходимые расширения. Цель такой инструкции — дать возможность другим разработчикам запустить проект у себя.
5. Планы по доработке проекта, если они есть. Например: «исправить X с помощью Y, чтобы получить Z». Если планов по доработке нет, не пишите о планах вовсе. Лучше ничего, чем абстрактное «сделать рефакторинг». В общем, чем конкретнее — тем лучше.