Методическое пособие:

Проектирование и разработка веб-приложений. Разработка на стороне клиента

Автор: Грушевский Юрий Викторович

Контакты: jurapro@yandex.ru

Оглавление

[**1. Введение**](#_860qtefy4d9v) **7**

[Для кого предназначено данное методическое пособие](#_fllkl791hghn) 7

[Необходимые начальные знания](#_8pofxy8nubd) 7

[**2. Основы JavaScript**](#_l7shqrmrj21v) **8**

[Введение в JavaScript](#_pqciw8rowujc) 8

[Подключение скриптов](#_l1jl6bysa041) 9

[Практическая работа №1](#_t1ujrf1z0lvg) 10

[Структура кода](#_uh1catta7h22) 10

[Инструкции](#_4zuypaon9x8y) 10

[Комментарии](#_qp9oj3xd40o6) 10

[Переменные](#_2e7ss25l1jij) 11

[Практическая работа №2](#_hvym9kowpfub) 13

[Константы](#_3krq21olo42j) 13

[Типы данных](#_daf7802ia13r) 14

[Число](#_iq772s8hbulu) 14

[Строка](#_jrrnsbgldyr5) 15

[Логический тип](#_wb3x61640hf9) 16

[Значение «null»](#_wcflcg1epzux) 16

[Значение «undefined»](#_2vp7mfgq3dwo) 16

[Преобразование типов](#_jss1t1gii2hp) 17

[Строковое преобразование](#_rgqcsb28tvpy) 17

[Численное преобразование](#_bttqgp3jebne) 17

[Логическое преобразование](#_mycsuu8pxwt3) 18

[Операторы](#_1sw6hswqinl3) 18

[Термины: «унарный», «бинарный», «операнд»](#_fnmdjykckpev) 18

[Сложение строк, бинарный +](#_xhfi7kfbg6h3) 19

[Преобразование к числу, унарный плюс +](#_l196l6tw1c60) 19

[Приоритет операторов](#_z04eazhoi1aj) 19

[Присваивание](#_ed7lbtmy0bpf) 20

[Остаток от деления %](#_q0awdmbfm4lu) 20

[Возведение в степень \*\*](#_8yl0vtrtotvd) 20

[Инкремент/декремент](#_m1k5gwsk3372) 21

[Операторы сравнения](#_dnapfnxg8qyw) 21

[Сравнение строк](#_9ybg1yw8q3jr) 21

[Сравнение разных типов](#_yfj4mp1yfw0t) 22

[Строгое сравнение](#_6pcg8i2kvw7) 22

[Взаимодействие: alert, prompt, confirm](#_5lskdxazaszj) 22

[alert](#_qrnpk8l376kb) 22

[prompt](#_xwazxbg95m03) 22

[confirm](#_8dnqpxrct5gl) 23

[Практическая работа №3](#_9n0szr76lz5x) 23

[Условные операторы: if, '?'](#_pgyj5swm9yzm) 24

[Практическая работа №4](#_88pphirgaqvv) 25

[Логические операторы](#_o5mig6blum72) 26

[|| (ИЛИ)](#_v589aj886y2w) 26

[&& (И)](#_ya1wzeourebz) 27

[! (НЕ)](#_f9niv0iiczzn) 28

[Практическая работа №5](#_fhy9pnavis0i) 28

[Циклы while и for](#_k7mh3fgpvk5s) 29

[Цикл «while»](#_qcrekkfchvfb) 29

[Цикл «do…while»](#_k72zar2i01iy) 30

[Цикл «for»](#_3aqyxu4h8ryw) 30

[Прерывание цикла: «break»](#_sna4ussytiqy) 30

[Переход к следующей итерации: continue](#_mw3tcs6rdulb) 31

[Практическая работа №6](#_rjd6q7wbwgkn) 31

[Конструкция "switch"](#_6sca9pm2uwr) 32

[Практическая работа №7](#_3km09nxaqlbq) 33

[Функции](#_dhqxafhgigkz) 33

[Объявление функции](#_808mjxza5byc) 34

[Локальные переменные](#_al94sfnj7aap) 35

[Внешние переменные](#_sx788m5m5jdx) 35

[Параметры](#_dkj64rcagvlt) 36

[Параметры по умолчанию](#_niw4deq5wcy8) 37

[Возврат значения](#_z4wmnynw6xlx) 37

[Выбор имени функции](#_4jxb8zgn2laz) 38

[Практическая работа №8](#_nti6vxh6tz0g) 39

[Function Expression](#_k6950pnkyz1) 39

[Функции-«колбэки»](#_40dge5b3f5f3) 39

[Функции-стрелки, основы](#_whecd4ehgwyv) 41

[Практическая работа №9](#_y9qzsikgnexx) 42

[Проверочная работа №1](#_qynaahnb5lnk) 43

[Задание №1 (по вариантам)](#_i7s5bqtsy90g) 43

[Задание №2 (по вариантам)](#_ted5lip9c7zv) 44

[Задание №3 (по вариантам)](#_cxsohxgmd6rv) 45

[Критерии оценивания](#_rjs213yrbyp) 46

[**3. Основы работы с объектами**](#_s1r2x42omyez) **47**

[Объекты](#_f7qvd5b5volp) 47

[Литералы и свойства](#_t2fqnpjvdf9t) 48

[Квадратные скобки](#_pvfrnb4p8p1e) 50

[Вычисляемые свойства](#_jcmre9p1ugpc) 51

[Свойство из переменной](#_hu4bvssdju4f) 51

[Проверка существования свойства](#_gnijuu9lutqq) 52

[Цикл «for…in»](#_sx6jtx5aglyy) 53

[Копирование по ссылке](#_ev0k6knblh2e) 53

[Сравнение объектов](#_8f5r4ihugmd8) 55

[Клонирование и объединение объектов, Object.assign](#_wyifoqk9kvcu) 56

[Практическая работа №10](#_5cwm11h2wz03) 58

[Методы объекта, "this"](#_dwvxi79dtnvx) 59

[Примеры методов](#_6v0spyhj349c) 59

[Сокращённая запись метода](#_hpbve0skp54y) 59

[Ключевое слово «this» в методах](#_ywi3ec27bd1j) 60

[«this» не является фиксированным](#_jyh1h271rymc) 61

[У стрелочных функций нет «this»](#_cdnikgh4qqbg) 62

[Практическая работа №11](#_l23q0q5c61mz) 62

[Преобразование объектов в примитивы](#_xvj56upic3fz) 63

[Преобразование к примитивам](#_t9ivsx8je4k8) 64

[Symbol.toPrimitive](#_corrwq9h6h0w) 65

[Методы toString/valueOf](#_s189phw1mwn) 66

[Возвращаемые типы](#_uj6tdy63ve8i) 67

[Конструкторы, создание объектов через "new"](#_ftkiy0kebm2h) 68

[Функция-конструктор](#_yg91s1f28ykl) 68

[Создание методов в конструкторе](#_nwf1s252t6nr) 69

[Практическая работа №12](#_83snnxck3l89) 70

[**4. Основы работы с массивами**](#_gfyfzr3aw1po) **71**

[Массивы](#_ftaj5armfawh) 71

[Объявление](#_v048npgbuoey) 71

[Методы pop/push, shift/unshift](#_7gn5rbohwz0i) 72

[Перебор элементов](#_ocng3gkp2z2x) 75

[Многомерные массивы](#_xdt4o7isqwrv) 75

[toString](#_mu097l38jnw6) 75

[Практическая работа №13](#_xrly3lwzilke) 76

[Методы массивов](#_o2tktd4c5qug) 76

[Добавление/удаление элементов](#_ef3nxmqk21ja) 76

[splice](#_2ows9m88tk4v) 76

[slice](#_llwbv1a7d3ek) 78

[concat](#_yddnvnmpalg4) 78

[Перебор: forEach](#_v4ge0z76gfrr) 79

[Поиск в массиве](#_14ggu1qomaq5) 79

[indexOf/lastIndexOf и includes](#_iq2ypd9bvpw1) 79

[find и findIndex](#_qrzimdu10ge) 80

[filter](#_f3cb79jcviql) 81

[Преобразование массива](#_tlces27iwt92) 81

[map](#_p3b6rey3exwx) 81

[sort(fn)](#_bqyjcxfykl65) 82

[reverse](#_lfi44vwrbopb) 83

[split и join](#_movxlsw3l26d) 84

[Практическая работа №14](#_s2yypg6596hp) 84

[Проверочная работа №2](#_wgeznjnod44) 86

[Обобщенное задание](#_duye3ks2w5l) 86

[Варианты](#_24g0bh1lfsi0) 86

[Критерии оценивания](#_8bwaj3nwtyy) 87

[**5. Документ и объекты страницы**](#_twacxng77bvv) **88**

[Браузерное окружение](#_v3xs0258jipf) 88

[DOM (Document Object Model)](#_evtrztrihyde) 89

[BOM (Browser Object Model)](#_yk16x43pjqqn) 89

[DOM-дерево](#_uxdg0690k0ca) 90

[Пример DOM](#_27rkwsqwjsgu) 90

[Другие типы узлов](#_nqiephrb5jlj) 91

[Навигация по DOM-элементам](#_qzzx53igdxmi) 93

[Сверху: documentElement и body](#_yqo0s3iie12y) 93

[Дети: childNodes, firstChild, lastChild](#_grxckxls5sw) 94

[DOM-коллекции](#_ssdukeyhnxy2) 95

[Соседи и родитель](#_gymbsgagiomh) 95

[Навигация только по элементам](#_itiussoip0fv) 95

[Практическая работа №15](#_c4nx7qhqkmib) 96

[Поиск: getElement\*, querySelector\*](#_629kv0po32u7) 97

[document.getElementById или просто id](#_k0biy3ostkgg) 97

[querySelectorAll](#_i0ij3eo9or3x) 98

[querySelector](#_pylgiw8p6mza) 98

[getElementsBy\*](#_jzvccycte6lx) 98

[Свойства узлов: тип, тег и содержимое](#_ats1apdaqzvf) 99

[Классы DOM-узлов](#_pbdwcqvg2wp) 99

[Тег: nodeName и tagName](#_9mave6c1av5f) 101

[innerHTML: содержимое элемента](#_6p7vdtdwqstf) 101

[outerHTML: HTML элемента целиком](#_e4eymixtduoa) 102

[nodeValue/data: содержимое текстового узла](#_dfhbossog3zj) 102

[textContent: просто текст](#_2oa22ybjq7xn) 103

[Свойство «hidden»](#_exboneaskrho) 103

[Другие свойства](#_fxyrlkw56173) 104

[Практическая работа №16](#_7usyde6yzr3) 104

[Атрибуты и свойства](#_zduz44lxydfl) 105

[DOM-свойства](#_mdj73nd4gyn5) 105

[HTML-атрибуты](#_vbe4wodv2ugj) 105

[Синхронизация между атрибутами и свойствами](#_shufv1lay464) 106

[DOM-свойства типизированы](#_n3bziuh5hmzz) 106

[Практическая работа №17](#_dsp7bl238l43) 106

[Изменение документа](#_p9vncg9m2s57) 107

[Пример: показать сообщение](#_5oedl91xx22o) 107

[Создание элемента](#_67h0mu8ax302) 108

[Методы вставки](#_oo10wz81hjn4) 108

[Удаление узлов](#_h16ycendkhnm) 109

[Клонирование узлов: cloneNode](#_7mxbh2od11k9) 110

[Практическая работа №18](#_izi2y4vtwe54) 111

[Стили и классы](#_b2brimg1ent4) 111

[className и classList](#_uks3ufcr1cjt) 112

[Element style](#_x43otwigcsx9) 113

[Практическая работа №19](#_sxtcz92wsz6u) 113

[Размеры и прокрутка элементов](#_n7ndmkyspkoo) 114

[Метрики](#_m3g3b94k3dn3) 115

[offsetParent, offsetLeft/Top](#_3ylzqppkfext) 115

[offsetWidth/Height](#_w6nf406rcaa8) 116

[clientTop/Left](#_nik6639nvm2g) 117

[clientWidth/Height](#_utbgg61an9f2) 118

[scrollWidth/Height](#_iotkwmit81jv) 119

[scrollLeft/scrollTop](#_o16suseezer7) 120

[Координаты](#_iwf6sp8zpia9) 120

[Координаты относительно окна: getBoundingClientRect](#_auuvwlq55qk9) 120

[elementFromPoint(x, y)](#_dx3jfs4s08lj) 121

[Координаты относительно документа](#_lhzv6z8kz58f) 122

[Практическая работа №20](#_pp6n4rgaejtt) 122

[**6. Браузерные события**](#_n28itiy807tr) **124**

[Введение в браузерные события](#_z4z7rjkwzwjh) 124

[Обработчики событий](#_jz4lebkpxdk3) 124

[Использование атрибута HTML](#_l0q29rymbfo0) 125

[Использование свойства DOM-объекта](#_npf3fkle0r4a) 125

[Доступ к элементу через this](#_zd08n17bd6i2) 126

[addEventListener](#_rz4179m4tmu4) 126

[Объект события](#_j4eop21ndjqq) 127

[Действия браузера по умолчанию](#_4rgfy2wrs4fi) 128

[Отмена действия браузера](#_luydg94ulh79) 128

[Практическая работа №21](#_8wm47o7dqpv3) 129

[Основы событий мыши](#_yign60osmrng) 129

[Типы событий мыши](#_okntg7rjyij5) 130

[Простые события](#_ie03nia649cv) 130

[Комплексные события](#_1cil23pqury6) 130

[Порядок событий](#_vr7tfnsc1he1) 130

[Получение информации о кнопке: which](#_fax127atf3o2) 130

[Модификаторы: shift, alt, ctrl и meta](#_mq7qdwco9ld3) 131

[Координаты: clientX/Y, pageX/Y](#_2pbf6vu38nwx) 131

[Практическая работа №22](#_m72xrdqmnldt) 132

[Движение мыши](#_h2mk2ytehm2f) 132

[События mouseover/mouseout, relatedTarget](#_nuldiafqcijr) 132

[Пропуск элементов](#_pxaeqrgzvz59) 133

[Событие mouseout при переходе на потомка](#_hb94xy6x6s9) 133

[События mouseenter и mouseleave](#_i30zp5be49zv) 133

[Drag'n'Drop с событиями мыши](#_xw0pxtoyt32j) 134

[Алгоритм Drag’n’Drop](#_hy5ub43wdwb3) 134

[Цели переноса (droppable)](#_46mynn53nwsn) 136

[Практическая работа №23](#_bk95n57jf546) 138

[Клавиатура: keydown и keyup](#_pid14qu1o007) 138

[События keydown и keyup](#_1358343hydc5) 138

[Автоповтор](#_fdg6bcrtmqhl) 138

[Действия по умолчанию](#_n34m0egg5yk) 139

[Проверочная работа №3](#_h6bqt8rrixdi) 139

[Варианты](#_vwnm3avse4i9) 139

[Критерии оценивания](#_c4jc0o5epr19) 152

[**7. Использованные источники**](#_2llqj66mtpnh) **153**

# 

# 1. Введение

## Для кого предназначено данное методическое пособие

Данное пособие предназначено для знакомства с клиентской веб-разработкой. Представленный материал подойдет для самостоятельного изучения и использования как основы для организации учебных практик и практических занятий в учебных заведениях.

## Необходимые начальные знания

Предполагается, что читатель знаком с основами программирования, основами HTML и CSS.

# 2. Основы JavaScript

## Введение в JavaScript

Изначально JavaScript был создан, чтобы «сделать веб-страницы живыми».

Программы на этом языке называются *скриптами*. Они могут встраиваться в HTML и выполняться автоматически при загрузке веб-страницы.

Скрипты распространяются и выполняются, как простой текст. Им не нужна специальная подготовка или компиляция для запуска.

Сегодня JavaScript может выполняться не только в браузере, но и на сервере или на любом другом устройстве, которое имеет специальную программу, называющуюся «движком» JavaScript.

У браузера есть собственный движок, который иногда называют «виртуальная машина JavaScript».

Современный JavaScript – это «безопасный» язык программирования. Он не предоставляет низкоуровневый доступ к памяти или процессору, потому что изначально был создан для браузеров, не требующих этого.

Возможности JavaScript сильно зависят от окружения, в котором он работает. Например, Node.JS поддерживает функции чтения/записи произвольных файлов, выполнения сетевых запросов и т.д.

В браузере для JavaScript доступно всё, что связано с манипулированием веб-страницами, взаимодействием с пользователем и веб-сервером.

Например, в браузере JavaScript может:

* Добавлять новый HTML-код на страницу, изменять существующее содержимое, модифицировать стили.
* Реагировать на действия пользователя, щелчки мыши, перемещения указателя, нажатия клавиш.
* Отправлять сетевые запросы на удалённые сервера, скачивать и загружать файлы (технологии AJAX и COMET).
* Получать и устанавливать куки, задавать вопросы посетителю, показывать сообщения.
* Запоминать данные на стороне клиента («local storage»).

Возможности JavaScript в браузере ограничены ради безопасности пользователя. Цель заключается в предотвращении доступа недобросовестной веб-страницы к личной информации или нанесения ущерба данным пользователя.

Примеры таких ограничений включают в себя:

* JavaScript на веб-странице не может читать/записывать произвольные файлы на жёстком диске, копировать их или запускать программы. Он не имеет прямого доступа к системным функциям ОС.
* Современные браузеры позволяют ему работать с файлами, но с ограниченным доступом, и предоставляют его, только если пользователь выполняет определенные действия, такие как «перетаскивание» файла в окно браузера или его выбор с помощью тега <input>.
* Существуют способы взаимодействия с камерой/микрофоном и другими устройствами, но они требуют явного разрешения пользователя. Таким образом, страница с поддержкой JavaScript не может незаметно включить веб-камеру, наблюдать за происходящим и отправлять информацию в ФСБ.
* Различные окна/вкладки не знают друг о друге. Иногда одно окно, используя JavaScript, открывает другое окно. Но даже в этом случае JavaScript с одной страницы не имеет доступа к другой, если они пришли с разных сайтов (с другого домена, протокола или порта).
* JavaScript может легко взаимодействовать с сервером, с которого пришла текущая страница. Но его способность получать данные с других сайтов/доменов ограничена. Хотя это возможно в принципе, для чего требуется явное согласие (выраженное в заголовках HTTP) с удалённой стороной. Опять же, это ограничение безопасности.

Как минимум, три сильные стороны JavaScript:

* Полная интеграция с HTML/CSS.
* Простые вещи делаются просто.
* Поддерживается всеми основными браузерами и включён по умолчанию.

## Подключение скриптов

Программы на JavaScript могут быть вставлены в любое место HTML-документа с помощью тега <script> (Листинг 2.1)

**Листинг 2.1** Включение скрипта в тело страницы

<!DOCTYPE HTML>

<html>

<body>

<p>Перед скриптом...</p>

<script>

*alert*( **'Привет, мир!'** );

</script>

<p>...После скрипта.</p>

</body>

</html>

Если у вас много JavaScript-кода, вы можете поместить его в отдельный файл.

Файл скрипта можно подключить к HTML с помощью атрибута src (Листинг 2.2).

**Листинг 2.2** Подключение скрипта отдельным файлом

<script **src**=**"/path/to/script.js"**></script>

### Практическая работа №1

1. Создайте страницу, которая отобразит сообщение «Я JavaScript!»;
2. Возьмите решение предыдущей задачи Вызвать alert, и измените его. Извлеките содержимое скрипта во внешний файл alert.js, лежащий в той же папке. Откройте страницу, убедитесь, что оповещение работает.

## Структура кода

### Инструкции

*Инструкции* – это синтаксические конструкции и команды, которые выполняют действия.

Мы уже видели инструкцию alert('Привет, мир!'), которая отображает сообщение «Привет, мир!».

В нашем коде может быть столько инструкций, сколько мы захотим. Инструкции могут отделяться точкой с запятой.

### Комментарии

Со временем программы становятся всё сложнее и сложнее. Возникает необходимость добавлять комментарии, которые бы описывали, что делает код и почему.

Комментарии могут находиться в любом месте скрипта. Они не влияют на его выполнение, поскольку движок просто игнорирует их.

**Однострочные комментарии начинаются с двойной косой черты //.**

Часть строки после // считается комментарием. Такой комментарий может как занимать строку целиком, так и находиться после инструкции (Листинг 2.3).

**Листинг 2.3** Однострочный комментарий

*// Этот комментарий занимает всю строку*

*alert*(**'Привет'**);

*alert*(**'Мир'**); *// Этот комментарий следует за инструкцией*

**Многострочные комментарии начинаются косой чертой со звёздочкой /\* и заканчиваются звёздочкой с косой чертой \*/** (Листинг 2.4).

**Листинг 2.4** Многострочный комментарий

*/\* Пример с двумя сообщениями.*

*Это - многострочный комментарий.*

*\*/*

*alert*(**'Привет'**);

*alert*(**'Мир'**);

## Переменные

*Переменная* – это «именованное хранилище» для данных. Мы можем использовать переменные для хранения товаров, посетителей и других данных.

Для создания переменной в JavaScript используйте ключевое слово let (Листинг 2.5).

**Листинг 2.5** Объявление переменной

**let *message***;

Теперь можно поместить в нее данные, используя оператор присваивания = (Листинг 2.6).

**Листинг 2.6** Присваивание переменной

**let *message***;

***message*** = **'Hello'**; *// сохранить строку*

Строка сохраняется в области памяти, связанной с переменной. Мы можем получить к ней доступ, используя имя переменной (Листинг 2.7).

**Листинг 2.7** Использование переменной

**let *message***;

***message*** = **'Hello!'**;

*alert*(***message***); *// показывает содержимое переменной*

Мы легко поймём концепцию «переменной», если представим её в виде «коробки» для данных с уникальным названием на ней.

Например, переменную message можно представить как коробку с названием "message" и значением "Hello!" внутри (Рисунок 2.1).



**Рисунок. 2.1.** Визуализация переменной

Мы можем положить любое значение в коробку.

Мы также можем изменить его столько раз, сколько захотим (Листинг 2.8)

**Листинг 2.8** Изменение значение переменной

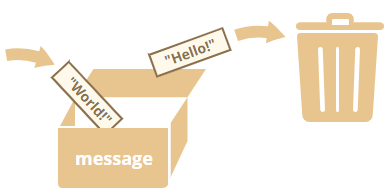
**let *message***;

***message*** = **'Hello!'**;

***message*** = **'World!'**; *// значение изменено*

*alert*(***message***);

При изменении значения старые данные удаляются из переменной (Рисунок 2.2).



**Рисунок. 2.2.** Визуализация изменения значения переменной

Мы также можем объявить две переменные и скопировать данные из одной в другую (Листинг 2.9).

**Листинг 2.9** Копирование значение одной переменной в другую

**let *hello*** = **'Hello world!'**;

**let *message***;

*// копируем значение 'Hello world' из переменной hello в переменную message*

***message*** = ***hello***;

*// теперь две переменные содержат одинаковые данные*

*alert*(***hello***); *// Hello world!*

*alert*(***message***); *// Hello world!*

В JavaScript есть два ограничения, касающиеся имен переменных:

1. Имя переменной должно содержать только буквы, цифры или символы $ и \_.
2. Первый символ не должен быть цифрой.

### Практическая работа №2

1. Работа с переменными
   * Объявите две переменные: admin и name.
   * Запишите строку "Джон" в переменную name.
   * Скопируйте значение из переменной name в admin.
   * Выведите на экран значение admin, используя функцию alert (должна показать «Джон»).
2. Именование переменных
   * Создайте переменную для названия нашей планеты. Как бы вы её назвали?
   * Создайте переменную для хранения имени текущего посетителя сайта. Как бы вы назвали такую переменную?

## Константы

Чтобы объявить константную, то есть, неизменяемую переменную, используйте const вместо let (Листинг 2.10).

**Листинг 2.10** Объявление констант

**const *myBirthday*** = **'18.04.1982'**;

Переменные, объявленные с помощью const, называются «константами». Их нельзя изменить. Попытка сделать это приведёт к ошибке.

Широко распространена практика использования констант в качестве псевдонимов для трудно запоминаемых значений, которые известны до начала исполнения скрипта.

Названия таких констант пишутся с использованием заглавных букв и подчеркивания (Листинг 2.11).

**Листинг 2.11** Именование констант

**const *COLOR\_RED*** = **"#F00"**;

**const *COLOR\_GREEN*** = **"#0F0"**;

**const *COLOR\_BLUE*** = **"#00F"**;

**const *COLOR\_ORANGE*** = **"#FF7F00"**;

*// ...когда нам нужно выбрать цвет*

**let *color*** = ***COLOR\_ORANGE***;

*alert*(***color***); *// #FF7F00*

## Типы данных

Переменная в JavaScript может содержать любые данные. В один момент там может быть строка, а в другой – число (Листинг 2.12).

**Листинг 2.12** Изменение типа переменной

*// Не будет ошибкой*

**let *message*** = **"hello"**;

***message*** = 123456;

Языки программирования, в которых такое возможно, называются «динамически типизированными». Это значит, что типы данных есть, но переменные не привязаны ни к одному из них.

### Число

*Числовой тип* данных (number) представляет как целочисленные значения, так и числа с плавающей точкой.

Существует множество операций для чисел, например, умножение \*, деление /, сложение +, вычитание - и так далее.

Кроме обычных чисел, существуют так называемые «специальные числовые значения», которые относятся к этому типу данных: Infinity, -Infinity и NaN.

Infinity представляет собой математическую бесконечность ∞. Это особое значение, которое больше любого числа.

NaN означает вычислительную ошибку. Это результат неправильной или неопределённой математической операции.

### Строка

Строка (string) в JavaScript должна быть заключена в кавычки (Листинг 2.13).

**Листинг 2.13** Инициализация строковой переменной

**let *str*** = **"Привет"**;

**let *str2*** = **'Одинарные кавычки тоже подойдут'**;

**let *phrase*** = **`Обратные кавычки позволяют встраивать переменные** ${***str***}**`**;

В JavaScript существует три типа кавычек.

1. Двойные кавычки: "Привет".
2. Одинарные кавычки: 'Привет'.
3. Обратные кавычки: `Привет`.

Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript.

Обратные кавычки же имеют «расширенный функционал». Они позволяют нам встраивать выражения в строку, заключая их в ${…}.Пример в листинге 2.14.:

**Листинг 2.14** Использование обратных кавычек

**let *name*** = **"Иван"**;

*// Вставим переменную*

*alert*( **`Привет,** ${***name***}**!`** ); *// Привет, Иван!*

*// Вставим выражение*

*alert*( **`результат:** ${1 + 2}**`** ); *// результат: 3*

Выражение внутри ${…} вычисляется, и его результат становится частью строки. Мы можем положить туда всё, что угодно: переменную name или выражение 1 + 2, или что-то более сложное.

Обратите внимание, что это можно делать только в обратных кавычках. Другие кавычки не имеют такого функционала встраивания!

### Логический тип

*Булевый тип* (boolean) может принимать только два значения: true (истина) и false (ложь). Такой тип, как правило, используется для хранения значений да/нет: true значит «да, правильно», а false значит «нет, не правильно» (Листинг 2.15).

**Листинг 2.15** Логический тип данных

**let *nameFieldChecked*** = **true**; *// да, поле отмечено*

**let *ageFieldChecked*** = **false**; *// нет, поле не отмечено*

Булевые значения также могут быть результатом сравнений (Листинг 2.16)

**Листинг 2.16** Использование логического типа

**let *isGreater*** = 4 > 1;

*alert*( ***isGreater*** ); *// true (результатом сравнения будет "да")*

### Значение «null»

Специальное значение null не относится ни к одному из типов, описанных выше.

Оно формирует отдельный тип, который содержит только значение null (Листинг 2.17).

**Листинг 2.17** Тип данных null

**let *age*** = **null**;

В JavaScript null не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках.

Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно».

### Значение «undefined»

Специальное значение undefined также стоит особняком. Оно формирует тип из самого себя так же, как и null.

Оно означает, что «значение не было присвоено».

Если переменная объявлена, но ей не присвоено никакого значения, то её значением будет undefined (Листинг 2.18).

**Листинг 2.18** Тип данных undefined

**let *x***;

*alert*(***x***); *// выведет "undefined"*

## Преобразование типов

Чаще всего операторы и функции автоматически приводят переданные им значения к нужному типу.

Например, alert автоматически преобразует любое значение к строке. Математические операторы преобразуют значения к числам.

Есть также случаи, когда нам нужно явно преобразовать значение в ожидаемый тип.

### Строковое преобразование

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки.

Например, alert(value) преобразует значение к строке.

Также мы можем использовать функцию String(value), чтобы преобразовать значение к строке (Листинг 2.19).

**Листинг 2.19** Строковое преобразование

**let *value*** = **true**;

*alert*(**typeof *value***); *// boolean*

***value*** = ***String***(***value***); *// теперь value это строка "true"*

*alert*(**typeof *value***); *// string*

Преобразование происходит очевидным образом. false становится "false", null становится "null" и т.п.

### Численное преобразование

Численное преобразование происходит в математических функциях и выражениях.

Например, когда операция деления / применяется не к числу (Листинг 2.20)

**Листинг 2.20** Численное преобразование

*alert*( **"6"** / **"2"** ); *// 3, Строки преобразуются в числа*

Мы можем использовать функцию Number(value), чтобы явно преобразовать value к числу (Листинг 2.21)

**Листинг 2.21** Численное преобразование

**let *str*** = **"123"**;

*alert*(**typeof *str***); *// string*

**let *num*** = ***Number***(***str***); *// становится числом 123*

*alert*(**typeof *num***); *// number*

Правила численного преобразования:

|  |  |
| --- | --- |
| **Значение** | **Преобразуется в…** |
| undefined | NaN |
| null | 0 |
| true / false | 1 / 0 |
| string | Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то 0, иначе из непустой строки «считывается» число. При ошибке результат NaN. |

### Логическое преобразование

Логическое преобразование самое простое.

Происходит в логических операторах (позже мы познакомимся с подобными конструкциями), но также может быть выполнено явно с помощью функции Boolean(value).

Правило преобразования:

* Значения, которые интуитивно «пустые», вроде 0, пустой строки, null, undefined и NaN, становятся false.
* Все остальные значения становятся true.

## Операторы

Многие операторы знакомы нам ещё со школы: сложение +, умножение \*, вычитание - и так далее.

В этой главе мы сконцентрируемся на операторах, которые в курсе математики не проходят.

### Термины: «унарный», «бинарный», «операнд»

*Операнд* – то, к чему применяется оператор. Например, в умножении 5 \* 2 есть два операнда: левый операнд равен 5, а правый операнд равен 2. Иногда их называют «аргументами» вместо «операндов».

*Унарным* называется оператор, который применяется к одному операнду. Например, оператор унарный минус "-" меняет знак числа на противоположный.

*Бинарным* называется оператор, который применяется к двум операндам. Тот же минус существует и в бинарной форме.

### Сложение строк, бинарный +

Обычно при помощи плюса '+' складывают числа.

Но если бинарный оператор '+' применить к строкам, то он их объединяет в одну (Листинг 2.22)

**Листинг 2.22** Сложение строк

**let *s*** = **"моя"** + **"строка"**;

*alert*(***s***); *// моястрока*

### Преобразование к числу, унарный плюс +

Плюс + существует в двух формах: бинарной, которую мы использовали выше, и унарной.

Унарный, то есть примененный к одному значению, плюс + ничего не делает с числами. Но если операнд не число, унарный плюс преобразует его в число (Листинг 2.23).

**Листинг 2.23** Унарный +

*// Не влияет на числа*

**let *x*** = 1;

*alert*( +***x*** ); *// 1*

**let *y*** = -2;

*alert*( +***y*** ); *// -2*

*// Преобразует нечисла в числа*

*alert*( +**true** ); *// 1*

*alert*( +**""** ); *// 0*

### Приоритет операторов

В том случае, если в выражении есть несколько операторов – порядок их выполнения определяется *приоритетом*, или, другими словами, существует определенный порядок выполнения операторов.

Из школы мы знаем, что умножение в выражении 2 \* 2 + 1 выполняется раньше сложения. Это как раз и есть «приоритет». Говорят, что умножение имеет более высокий приоритет, чем сложение.

Скобки важнее, чем приоритет, так что если мы не удовлетворены порядком по умолчанию, мы можем использовать их, чтобы изменить приоритет. Например, написать (1 + 2) \* 2.

В JavaScript много операторов. Каждый оператор имеет соответствующий номер приоритета. Тот, у кого это число больше – выполнится раньше. Если приоритет одинаковый, то порядок выполнения – слева направо.

Общую таблицу приоритетов операторов можно посмотреть здесь: <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Operator_Precedence>

### Присваивание

Давайте отметим, что в таблице приоритетов также есть оператор присваивания =. У него один из самых низких приоритетов.

Именно поэтому, когда переменной что-либо присваивают, например, x = 2 \* 2 + 1, то сначала выполнится арифметика, а уже затем происходит присваивание =.

**Оператор "=" возвращает значение.**

Все операторы возвращают значение. Для некоторых это очевидно, например сложение + или умножение \*. Но и оператор присваивания не является исключением.

Вызов x = value записывает value в x и возвращает его.

### Остаток от деления %

Оператор взятия остатка %, несмотря на обозначение, никакого отношения к процентам не имеет.

Его результат a % b – это остаток от деления a на b (Листинг 2.24).

**Листинг 2.24** Оператор %

*alert*( 5 % 2 ); *// 1, остаток от деления 5 на 2*

*alert*( 8 % 3 ); *// 2, остаток от деления 8 на 3*

*alert*( 6 % 3 ); *// 0, остаток от деления 6 на 3*

### Возведение в степень \*\*

Оператор возведения в степень \*\* недавно добавили в язык.

Для натурального числа b результат a \*\* b равен a, умноженному на само себя b раз.

### Инкремент/декремент

Одной из наиболее частых операций в JavaScript, как и во многих других языках программирования, является увеличение или уменьшение переменной на единицу.

Для этого существуют даже специальные операторы:

* Инкремент ++ увеличивает на 1;
* Декремент -- уменьшает на 1.

Инкремент/декремент можно применить только к переменной. Попытка использовать его на значении, типа 5++, приведёт к ошибке.

## Операторы сравнения

Многие операторы сравнения известны нам из математики:

* Больше/меньше: a > b, a < b.
* Больше/меньше или равно: a >= b, a <= b.
* Равно: a == b. Обратите внимание, для сравнения используется двойной знак равенства =. Один знак равенства a = b означал бы присваивание.
* Не равно. В математике обозначается символом ≠. В JavaScript записывается как знак равенства с предшествующим ему восклицательным знаком: a != b.

Операторы сравнения, как и другие операторы, возвращают значение. Это значение имеет логический тип:

* true – означает «да», «верно», «истина».
* false – означает «нет», «неверно», «ложь».

**Листинг 2.25** Возврат значения логического оператора

*alert*( 2 > 1 ); *// true (верно)*

*alert*( 2 == 1 ); *// false (неверно)*

*alert*( 2 != 1 ); *// true (верно)*

### Сравнение строк

Чтобы определить, что одна строка больше другой, JavaScript использует «алфавитный» или «лексикографический» порядок.

Другими словами, строки сравниваются посимвольно.

Алгоритм сравнения двух строк довольно прост:

1. Сначала сравниваются первые символы строк.
2. Если первый символ первой строки больше (меньше), чем первый символ второй, то первая строка больше (меньше) второй.
3. Если первые символы равны, то таким же образом сравниваются уже вторые символы строк.
4. Сравнение продолжается, пока не закончится одна из строк.
5. Если обе строки заканчиваются одновременно, то они равны. Иначе, большей считается более длинная строка.

### Сравнение разных типов

При сравнении значений разных типов JavaScript приводит каждое из них к числу (Листинг 2.26).

**Листинг 2.26** Сравнение разных типов

*alert*( **'2'** > 1 ); *// true, строка '2' становится числом 2*

*alert*( **'01'** == 1 ); *// true, строка '01' становится числом 1*

### Строгое сравнение

Использование обычного сравнения == может вызывать проблемы. Например, оно не отличает 0 от false.

Это происходит из-за того, что операнды разных типов преобразуются оператором == к числу. В итоге, и пустая строка, и false становятся нулём.

Как же тогда отличать 0 от false?

**Оператор строгого равенства === проверяет равенство без приведения типов.**

Другими словами, если a и b имеют разные типы, то проверка a === b немедленно возвращает false без попытки их преобразования.

## Взаимодействие: alert, prompt, confirm

### alert

Синтаксис:

*alert(message);*

Этот код отобразит окно в браузере и приостановит дальнейшее выполнение скриптов до тех пор, пока пользователь не нажмет кнопку «OK».

Это небольшое окно с сообщением называется модальным окном. Понятие модальное означает, что пользователь не может взаимодействовать с интерфейсом остальной части страницы, нажимать на другие кнопки и т.д. до тех пор, пока взаимодействует с окном. В данном случае – пока не будет нажата кнопка «OK».

### prompt

Функция prompt принимает два аргумента:

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

Этот код отобразит модальное окно с текстом, полем для ввода текста и кнопками OK/Отмена.

**title**

Текст для отображения в окне.

**default**

Необязательный второй параметр, который устанавливает начальное значение в поле для текста в окне.

Пользователь может напечатать что-либо в поле ввода и нажать OK. Он также может отменить ввод нажатием на кнопку «Отмена» или нажав на клавишу Esc.

Вызов prompt вернёт текст, указанный в поле для ввода, или null, если ввод отменен пользователем (Листинг 2.27)

**Листинг 2.27** Использование prompt

**let *age*** = *prompt*(**'Сколько тебе лет?'**, 100);

*alert*(**`Тебе** ${***age***} **лет!`**); *// Тебе 100 лет!*

### confirm

Синтаксис:

*result = confirm(question);*

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

Результат true, если нажата кнопка OK. В других случаях – false (Листинг 2.28).

**Листинг 2.28** Использование confirm

**let *isBoss*** = *confirm*(**"Ты здесь главный?"**);

*alert*( ***isBoss*** ); *// true, если нажата OK*

### Практическая работа №3

1. Создайте страницу, которая спрашивает имя у пользователя и выводит его.

## Условные операторы: if, '?'

Иногда нам нужно выполнить различные действия в зависимости от условий.

Для этого мы можем использовать оператор if и условный оператор ?, который также называют «оператор вопросительный знак».

Оператор if(...) вычисляет условие в скобках и, если результат true, то выполняет блок кода (Листинг 2.29).

**Листинг 2.29** Использование оператора if

**let *year*** = *prompt*(**'В каком году появилась спецификация ECMAScript-2015?'**, **''**);

**if** (***year*** == 2015) *alert*( **'Вы правы!'** );

Если мы хотим выполнить более одной инструкции, то нужно заключить блок кода в фигурные скобки.

Рекомендуется использовать фигурные скобки {} всегда, когда вы используете оператор if, даже если выполняется только одна команда. Это улучшает читабельность кода.

Оператор if (…) вычисляет выражение в скобках и преобразует результат к логическому типу.

Оператор if может содержать необязательный блок «else» («иначе»). Выполняется, когда условие ложно (Листинг 2.30).

**Листинг 2.30** Использование блока else

**let *year*** = *prompt*(**'В каком году появилась спецификация ECMAScript-2015?'**, **''**);

**if** (***year*** == 2015) {

*alert*( **'Да вы знаток!'** );

} **else** {

*alert*( **'А вот и неправильно!'** ); *// любое значение, кроме 2015*

}

Иногда, нужно проверить несколько вариантов условия. Для этого используется блок else if (Листинг 2.31).

**Листинг 2.31** Использование блока else if

**let *year*** = *prompt*(**'В каком году появилась спецификация ECMAScript-2015?'**, **''**);

**if** (***year*** < 2015) {

*alert*( **'Это слишком рано...'** );

} **else if** (***year*** > 2015) {

*alert*( **'Это поздновато'** );

} **else** {

*alert*( **'Верно!'** );

}

Иногда, нам нужно назначить переменную в зависимости от условия (Листинг 2.32).

**Листинг 2.32** Задание переменной в зависимости от условия

**let *accessAllowed***;

**let *age*** = *prompt*(**'Сколько вам лет?'**, **''**);

**if** (***age*** > 18) {

***accessAllowed*** = **true**;

} **else** {

***accessAllowed*** = **false**;

}

*alert*(***accessAllowed***);

Так называемый «условный» оператор «вопросительный знак» позволяет нам сделать это более коротким и простым способом.

Оператор представлен знаком вопроса ?. Его также называют «тернарный», так как этот оператор, единственный в своем роде, имеет три аргумента.

Синтаксис:

*let result = условие ? значение1 : значение2;*

Сначала вычисляется условие: если оно истинно, тогда возвращается значение1, в противном случае – значение2 (Листинг 2.33).

**Листинг 2.33** Тернарный оператор

**let *accessAllowed*** = (age > 18) ? **true** : **false**;

### Практическая работа №4

1. Используя конструкцию if..else, напишите код, который будет спрашивать: «Какое «официальное» название JavaScript?» Если пользователь вводит «ECMAScript», то показать: «Верно!», в противном случае – отобразить: «Не знаете? ECMAScript!»;
2. Используя конструкцию if..else, напишите код, который получает число через prompt, а затем выводит в alert:

* 1, если значение больше нуля,
* -1, если значение меньше нуля,
* 0, если значение равно нулю.

Предполагается, что пользователь вводит только числа.

1. Перепишите if с использованием условного оператора '?':

**let *result***;

**if** (a + b < 4) {

***result*** = **'Мало'**;

} **else** {

***result*** = **'Много'**;

}

## Логические операторы

В JavaScript есть три логических оператора: || (ИЛИ), && (И) и ! (НЕ).

Несмотря на своё название, данные операторы могут применяться к значениям любых типов. Полученные результаты также могут иметь различный тип.

### || (ИЛИ)

Оператор «ИЛИ» выглядит как двойной символ вертикальной черты:

*result = a || b;*

Традиционно в программировании ИЛИ предназначено только для манипулирования булевыми значениями: в случае, если какой-либо из аргументов true, он вернёт true, в противоположной ситуации возвращается false (Листинг 2.34).

**Листинг 2.34** Использование логического ИЛИ

**let *hour*** = 9;

**if** (***hour*** < 10 || ***hour*** > 18) {

*alert*( **'Офис закрыт.'** );

}

Описанная выше логика соответствует традиционной. Теперь давайте поработаем с «дополнительными» возможностями JavaScript.

Расширенный алгоритм работает следующим образом.

При выполнении ИЛИ || с несколькими значениями:

*result = value1 || value2 || value3;*

Оператор || выполняет следующие действия:

* Вычисляет операнды слева направо.
* Каждый операнд конвертирует в логическое значение. Если результат true, останавливается и возвращает исходное значение этого операнда.
* Если все операнды являются ложными (false), возвращает последний из них.
* Значение возвращается в исходном виде, без преобразования.

Другими словами, цепочка ИЛИ "||" возвращает первое истинное значение или последнее, если такое значение не найдено (Листинг 2.35).

**Листинг 2.35** Расширенные возможности логического ИЛИ

*alert*( 1 || 0 ); *// 1*

*alert*( **true** || **'no matter what'** ); *// true*

*alert*( **null** || 1 ); *// 1 (первое истинное значение)*

*alert*( **null** || 0 || 1 ); *// 1 (первое истинное значение)*

*alert*( **undefined** || **null** || 0 ); *// 0 (поскольку все ложно, возвращается последнее значение)*

### && (И)

Оператор И пишется как два амперсанда &&:

*result = a && b;*

В традиционном программировании И возвращает true, если оба аргумента истинны, а иначе – false (Листинг 2.36).

**Листинг 2.36** Традиционное использование оператора И

**let *hour*** = 12;

**let *minute*** = 30;

**if** (***hour*** == 12 && ***minute*** == 30) {

*alert*( **'The time is 12:30'** );

}

При нескольких подряд операторах И:

*result = value1 && value2 && value3;*

Оператор && выполняет следующие действия:

* Вычисляет операнды слева направо.
* Каждый операнд преобразует в логическое значение. Если результат false, останавливается и возвращает исходное значение этого операнда.
* Если все операнды были истинными, возвращается последний.
* Другими словами, И возвращает первое ложное значение. Или последнее, если ничего не найдено.

Вышеуказанные правила схожи с поведением ИЛИ. Разница в том, что И возвращает первое *ложное* значение, а ИЛИ –  первое *истинное* (Листинг 2.37).

**Листинг 2.37** Расширенное использование оператора И

*// Если первый операнд истинный,*

*// И возвращает второй:*

*alert*( 1 && 0 ); *// 0*

*alert*( 1 && 5 ); *// 5*

*// Если первый операнд ложный,*

*// И возвращает его. Второй операнд игнорируется*

*alert*( **null** && 5 ); *// null*

*alert*( 0 && **"no matter what"** ); *// 0*

### ! (НЕ)

Оператор НЕ представлен восклицательным знаком !.

Синтаксис довольно прост:

*result = !value;*

Оператор принимает один аргумент и выполняет следующие действия:

Сначала приводит аргумент к логическому типу true/false.

Затем возвращает противоположное значение.

Приоритет НЕ ! является наивысшим из всех логических операторов, поэтому он всегда выполняется первым, перед && или ||.

### Практическая работа №5

1. Напишите условие if для проверки, что переменная age находится в диапазоне между 14 и 90 включительно. «Включительно» означает, что значение переменной age может быть равно 14 или 90.
2. Напишите код, который будет спрашивать логин с помощью prompt.

Если посетитель вводит «Админ», то prompt запрашивает пароль, если ничего не введено или нажата клавиша Esc – показать «Отменено», в противном случае отобразить «Я вас не знаю».

Пароль проверять так:

* Если введен пароль «Я главный», то выводить «Здравствуйте!»,
* Иначе – «Неверный пароль»,
* При отмене – «Отменено».

## Циклы while и for

При написании скриптов зачастую встаёт задача сделать однотипное действие много раз.

Например, вывести товары из списка один за другим. Или просто перебрать все числа от 1 до 10 и для каждого выполнить одинаковый код.

Для многократного повторения одного участка кода предусмотрены циклы.

### Цикл «while»

Цикл while имеет следующий синтаксис:

*while (condition) {*

*// код*

*// также называемый "телом цикла"*

*}*

Код из тела цикла выполняется, пока условие condition истинно.

Например, цикл ниже выводит i, пока i < 3 (Листинг 2.38)

**Листинг 2.38** Пример использования цикла while

**let *i*** = 0;

**while** (***i*** < 3) { *// выводит 0, затем 1, затем 2*

*alert*( ***i*** );

***i***++;

}

Одно выполнение тела цикла по-научному называется *итерация*. Цикл в примере выше совершает три итерации.

Если бы строка i++ отсутствовала в примере выше, то цикл бы повторялся (в теории) вечно. На практике, конечно, браузер не позволит такому случиться, он предоставит пользователю возможность остановить «подвисший» скрипт, а JavaScript на стороне сервера придётся «убить» процесс.

Любое выражение или переменная может быть условием цикла, а не только сравнение: условие while вычисляется и преобразуется в логическое значение.

### Цикл «do…while»

Проверку условия можно разместить под телом цикла, используя специальный синтаксис do..while:

*do {*

*// тело цикла*

*} while (condition);*

Цикл сначала выполнит тело, а затем проверит условие condition, и пока его значение равно true, он будет выполняться снова и снова.

Такая форма синтаксиса оправдана, если вы хотите, чтобы тело цикла выполнилось хотя бы один раз, даже если условие окажется ложным. На практике чаще используется форма с предусловием: while(…) {…}.

### Цикл «for»

Более сложный, но при этом самый распространённый цикл — цикл for.

Выглядит он так:

*for (начало; условие; шаг) {*

*// ... тело цикла ...*

*}*

Цикл ниже выполняет alert(i) для i от 0 до (но не включая) 3 (Листинг 2.39).

**Листинг 2.39** Пример использования цикла for

**for** (**let *i*** = 0; ***i*** < 3; ***i***++) { *// выведет 0, затем 1, затем 2*

*alert*(***i***);

}

То есть, *начало* выполняется один раз, а затем каждая итерация заключается в проверке условия, после которой выполняется *тело* и *шаг*.

### Прерывание цикла: «break»

Обычно цикл завершается при вычислении *условия* в false.

Но мы можем выйти из цикла в любой момент с помощью специальной директивы break.

Например, следующий код подсчитывает сумму вводимых чисел до тех пор, пока посетитель их вводит, а затем – выдает результат (Листинг 2.40).

**Листинг 2.40** Пример использования команды break

**let *sum*** = 0;

**while** (**true**) {

**let *value*** = +*prompt*(**"Введите число"**, **''**);

**if** (!***value***) **break**; *// (\*)*

***sum*** += ***value***;

}

*alert*( **'Сумма: '** + ***sum*** );

Директива break в строке (\*) полностью прекращает выполнение цикла и передаёт управление на строку за его телом, то есть на alert.

Вообще, сочетание «бесконечный цикл + break» – отличная штука для тех ситуаций, когда условие, по которому нужно прерваться, находится не в начале или конце цикла, а посередине.

### Переход к следующей итерации: continue

Директива continue – «облегчённая версия» break. При её выполнении цикл не прерывается, а переходит к следующей итерации (если условие все ещё равно true).

Ее используют, если понятно, что на текущем повторе цикла делать больше нечего (Листинг 2.41).

**Листинг 2.41** Пример использования команды continue

**for** (**let *i*** = 0; ***i*** < 10; ***i***++) {

*// если true, пропустить оставшуюся часть тела цикла*

**if** (***i*** % 2 == 0) **continue**;

*alert*(***i***); *// 1, затем 3, 5, 7, 9*

}

Для четных значений i, директива continue прекращает выполнение тела цикла и передает управление на следующую итерацию for (со следующим числом). Таким образом alert вызывается только для нечётных значений.

### Практическая работа №6

1. При помощи цикла for выведите чётные числа от 2 до 10;
2. Перепишите код, заменив цикл for на while, без изменения поведения цикла.

**for** (**let *i*** = 0; ***i*** < 3; ***i***++) {

*alert*( **`number** ${***i***}**!`** );

}

1. Напишите цикл, который предлагает prompt ввести число, большее 100. Если посетитель ввел другое число – попросить ввести еще раз, и так далее.

Цикл должен спрашивать число пока либо посетитель не введет число, большее 100, либо не нажмет кнопку Отмена (ESC).

Предполагается, что посетитель вводит только числа.

1. Натуральное число, большее 1, называется простым, если оно ни на что не делится, кроме себя и 1. Другими словами, n > 1 – простое, если при его делении на любое число кроме 1 и n есть остаток. Например, 5 это простое число, оно не может быть разделено без остатка на 2, 3 и 4. Напишите код, который выводит все простые числа из интервала от 2 до n.

## Конструкция "switch"

Конструкция switch заменяет собой сразу несколько if.

Она представляет собой более наглядный способ сравнить выражение сразу с несколькими вариантами.

Конструкция switch имеет один или более блок case и необязательный блок default.

Выглядит она так:

*switch(x) {*

*case 'value1': // if (x === 'value1')*

*...*

*[break]*

*case 'value2': // if (x === 'value2')*

*...*

*[break]*

*default:*

*...*

*[break]*

*}*

* Переменная x проверяется на строгое равенство первому значению value1, затем второму value2 и так далее.
* Если соответствие установлено – switch начинает выполняться от соответствующей директивы case и далее, до ближайшего break (или до конца switch).
* Если ни один case не совпал – выполняется (если есть) вариант default.

Пример использования можно увидеть в Листинге 2.42.

**Листинг 2.42** Пример использования конструкции switch

**let *a*** = 2 + 2;

**switch** (***a***) {

**case** 3:

*alert*( **'Маловато'** );

**break**;

**case** 4:

*alert*( **'В точку!'** );

**break**;

**case** 5:

*alert*( **'Перебор'** );

**break**;

**default**:

*alert*( **"Нет таких значений"** );

}

Здесь оператор switch последовательно сравнит a со всеми вариантами из case.

Сначала 3, затем – так как нет совпадения – 4. Совпадение найдено, будет выполнен этот вариант, со строки alert( 'В точку!' ) и далее, до ближайшего break, который прервет выполнение.

**Если break нет, то выполнение пойдёт ниже по следующим case, при этом остальные проверки игнорируются.**

### Практическая работа №7

1. Перепишите код с использованием одной конструкции switch:

**const *number*** = +*prompt*(**'Введите число между 0 и 3'**, **''**);

**if** (***number*** === 0) {

*alert*(**'Вы ввели число 0'**);

}

**if** (***number*** === 1) {

*alert*(**'Вы ввели число 1'**);

}

**if** (***number*** === 2 || ***number*** === 3) {

*alert*(**'Вы ввели число 2, а может и 3'**);

}

## Функции

Зачастую нам надо повторять одно и то же действие во многих частях программы.

Например, необходимо красиво вывести сообщение при приветствии посетителя, при выходе посетителя с сайта, ещё где-нибудь.

Чтобы не повторять один и тот же код во многих местах, придуманы функции. Функции являются основными «строительными блоками» программы.

Примеры встроенных функций вы уже видели – это alert(message), prompt(message, default) и confirm(question). Но можно создавать и свои.

### Объявление функции

Для создания функций мы можем использовать *объявление функции* (Листинг 2.43).

**Листинг 2.43** Пример объявления функции

**function** *showMessage*() {

*alert*( **'Всем привет!'** );

}

Вначале идёт ключевое слово function, после него имя функции, затем список параметров в круглых скобках через запятую (в вышеприведенном примере он пустой) и, наконец, код функции, также называемый «телом функции», внутри фигурных скобок.

*function имя(параметры) {*

*...тело...*

*}*

Наша новая функция может быть вызвана по её имени: showMessage() (Листинг 2.44).

**Листинг 2.44** Пример вызова функции

**function** *showMessage*() {

*alert*( **'Всем привет!'** );

}

*showMessage*();

*showMessage*();

Вызов showMessage() выполняет код функции. Здесь мы увидим сообщение дважды.

Этот пример явно демонстрирует одно из главных предназначений функций: избавление от дублирования кода.

Если понадобится поменять сообщение или способ его вывода – достаточно изменить его в одном месте: в функции, которая его выводит.

### Локальные переменные

Переменные, объявленные внутри функции, видны только внутри этой функции (Листинг 2.45).

**Листинг 2.45** Локальные переменные функции

***function*** *showMessage() {*

***let*** *message =* ***"Привет, я JavaScript!"****; // локальная переменная*

*alert( message );*

*}*

*showMessage(); // Привет, я JavaScript!*

*alert( message ); // <-- будет ошибка, т.к. переменная видна только внутри функции*

### Внешние переменные

У функции есть доступ к внешним переменным, Листинг 2.46.

**Листинг 2.46** Использование внешних переменных внутри функции

**let *userName*** = **'Вася'**;

**function** *showMessage*() {

**let** message = **'Привет, '** + ***userName***;

*alert*(message);

}

*showMessage*(); *// Привет, Вася*

Функция обладает полным доступом к внешним переменным и может изменять их значение.

Внешняя переменная используется, только если внутри функции нет такой локальной.

Если одноименная переменная объявляется внутри функции, тогда она перекрывает внешнюю. Например, в коде Листинга 2.47 функция использует локальную переменную userName. Внешняя будет проигнорирована

**Листинг 2.47** Игнорирование внешней переменной

**let *userName*** = **'Вася'**;

**function** *showMessage*() {

**let** userName = **"Петя"**; *// объявляем локальную переменную*

**let** message = **'Привет, '** + userName; *// Петя*

*alert*(message);

}

*// функция создаст и будет использовать свою собственную локальную переменную userName*

*showMessage*();

*alert*( ***userName*** ); *// Вася, не изменилась, функция не трогала внешнюю переменную*

Переменные, объявленные снаружи всех функций, такие как внешняя переменная userName в вышеприведенном коде – называются *глобальными*.

*Глобальные переменные* видимы для любой функции (если только их не перекрывают одноименные локальные переменные).

Желательно сводить использование глобальных переменных к минимуму. В современном коде обычно мало или совсем нет глобальных переменных. Хотя они иногда полезны для хранения важнейших «общепроектовых» данных.

### Параметры

Мы можем передать внутрь функции любую информацию, используя параметры (также называемые *аргументы функции*).

В нижеприведенном примере функции передаются два параметра: from и text (Листинг 2.48).

**Листинг 2.48** Параметры функции

**function** *showMessage*(from, text) { *// аргументы: from, text*

*alert*(from + **': '** + text);

}

*showMessage*(**'Аня'**, **'Привет!'**); *// Аня: Привет! (\*)*

*showMessage*(**'Аня'**, **"Как дела?"**); *// Аня: Как дела? (\*\*)*

Когда функция вызывается в строках (\*) и (\*\*), переданные значения копируются в локальные переменные from и text. Затем они используются в теле функции.

Вот ещё один пример: у нас есть переменная from, и мы передаем ее функции. Обратите внимание: функция изменяет значение from, но это изменение не видно снаружи. Функция всегда получает только копию значения (Листинг 2.49):

**Листинг 2.49** Параметры функции - локальные переменные

**function** *showMessage*(from, text) {

from = **'\*'** + from + **'\*'**; *// немного украсим "from"*

*alert*( from + **': '** + text );

}

**let *from*** = **"Аня"**;

*showMessage*(***from***, **"Привет"**); *// \*Аня\*: Привет*

*// значение "from" осталось прежним, функция изменила значение локальной переменной*

*alert*( ***from*** ); *// Аня*

### Параметры по умолчанию

Если параметр не указан, то его значением становится undefined.

Если мы хотим задать параметру значение по умолчанию, мы должны указать его после = (Листинг 2.50).

**Листинг 2.50** Параметры функции - значения по умолчанию

**function** *showMessage*(from, text = **"текст не добавлен"**) {

*alert*( from + **": "** + text );

}

*showMessage*(**"Аня"**); *// Аня: текст не добавлен*

Теперь, если параметр text не указан, его значением будет "текст не добавлен"

В данном случае "текст не добавлен" это строка, но на её месте могло бы быть и более сложное выражение, которое бы вычислялось и присваивалось при отсутствии параметра.

### Возврат значения

Функция может вернуть результат, который будет передан в вызвавший её код.

Простейшим примером может служить функция сложения двух чисел (Листинг 2.51).

**Листинг 2.51** Возврат значения функции

**function** *sum*(a, b) {

**return** a + b;

}

**let *result*** = *sum*(1, 2);

*alert*( ***result*** ); *// 3*

Директива return может находиться в любом месте тела функции. Как только выполнение доходит до этого места, функция останавливается, и значение возвращается в вызвавший ее код (присваивается переменной result выше).

Возможно использовать return и без значения. Это приведет к немедленному выходу из функции.

### Выбор имени функции

Функция – это действие. Поэтому имя функции обычно является глаголом. Оно должно быть простым, точным и описывать действие функции, чтобы программист, который будет читать код, получил верное представление о том, что делает функция.

Как правило, используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение. Обычно в командах разработчиков действуют соглашения, касающиеся значений этих префиксов.

Например, функции, начинающиеся с "show" обычно что-то показывают.

Функции, начинающиеся с…

* "get…" – возвращают значение,
* "calc…" – что-то вычисляют,
* "create…" – что-то создают,
* "check…" – что-то проверяют и возвращают логическое значение, и т.д.

Примеры таких имен:

showMessage(..) *// показывает сообщение*

getAge(..) *// возвращает возраст (в каком либо значении)*

calcSum(..) *// вычисляет сумму и возвращает результат*

createForm(..) *// создаёт форму (и обычно возвращает её)*

checkPermission(..) *// проверяет доступ, возвращая true/false*

Благодаря префиксам, при первом взгляде на имя функции становится понятным что делает её код, и какое значение она может возвращать.

Функции должны быть короткими и делать только что-то одно. Если это что-то большое, имеет смысл разбить функцию на несколько меньших. Иногда следовать этому правилу непросто, но это определённо хорошее правило.

Небольшие функции не только облегчают тестирование и отладку – само существование таких функций выполняет роль хороших комментариев!

### Практическая работа №8

1. Напишите функцию min(a,b), которая возвращает меньшее из чисел a и b.
2. Напишите функцию pow(x,n), которая возвращает x в степени n. Иначе говоря, умножает x на себя n раз и возвращает результат.

### Function Expression

Функция в JavaScript – это не магическая языковая структура, а особого типа значение.

Синтаксис, который мы использовали до этого, называется *Function Declaration* (Объявление Функции):

***function*** *sayHi() {*

*alert(* ***"Привет"*** *);*

*}*

Существует еще один синтаксис создания функций, который называется *Function Expression* (Функциональное Выражение).

Оно выглядит вот так:

**let** *sayHi* = **function**() {

*alert*( **"Привет"** );

};

В коде выше функция создается и явно присваивается переменной, как любое другое значение. По сути без разницы, как мы определили функцию, это просто значение, хранимое в переменной sayHi.

Мы можем скопировать функцию в другую переменную (Листинг 2.52).

**Листинг 2.52** Копирование функции

**function** *sayHi*() { *// (1) создаём*

*alert*( **"Привет"** );

}

**let *func*** = *sayHi*; *// (2) копируем*

***func***(); *// Привет // (3) вызываем копию (работает)!*

*sayHi*(); *// Привет // прежняя тоже работает (почему бы нет)*

### Функции-«колбэки»

Рассмотрим еще примеры функциональных выражений и передачи функции как значения.

Давайте напишем функцию ask(question, yes, no) с тремя параметрами:

**question**

Текст вопроса

**yes**

Функция, которая будет вызываться, если ответ будет «Yes»

**no**

Функция, которая будет вызываться, если ответ будет «No»

Наша функция должна задать вопрос question и, в зависимости от того, как ответит пользователь, вызвать yes() или no() (Листинг 2.53).

**Листинг 2.53** Функции как параметры другой функции

**function** *ask*(question, yes, no) {

**if** (*confirm*(question)) yes()

**else** no();

}

**function** *showOk*() {

*alert*( **"Вы согласны."** );

}

**function** *showCancel*() {

*alert*( **"Вы отменили выполнение."** );

}

*// использование: функции showOk, showCancel передаются в качестве аргументов ask*

*ask*(**"Вы согласны?"**, *showOk*, *showCancel*);

На практике подобные функции очень полезны. Основное отличие «реальной» функции ask от примера выше будет в том, что она использует более сложные способы взаимодействия с пользователем, чем простой вызов confirm. В браузерах такие функции обычно отображают красивые диалоговые окна.

**Аргументы функции ask ещё называют функциями-колбэками или просто колбэками.**

Ключевая идея в том, что мы передаём функцию и ожидаем, что она вызовется обратно (от англ. «call back» – обратный вызов) когда-нибудь позже, если это будет необходимо. В нашем случае, showOk становится колбэком для ответа «yes», а showCancel – для ответа «no».

Мы можем переписать этот пример значительно короче, используя Function Expression (Листинг 2.54).

**Листинг 2.54** Описание колбэков в Function Expression

**function** *ask*(question, yes, no) {

**if** (*confirm*(question)) yes()

**else** no();

}

*ask*(

**"Вы согласны?"**,

**function**() { *alert*(**"Вы согласились."**); },

**function**() { *alert*(**"Вы отменили выполнение."**); }

);

Здесь функции объявляются прямо внутри вызова ask(...). У них нет имён, поэтому они называются *анонимными*. Такие функции недоступны снаружи ask (потому что они не присвоены переменным), но это как раз то, что нам нужно.

Подобный код, появившийся в нашем скрипте выглядит очень естественно, в духе JavaScript.

### Функции-стрелки, основы

Существует ещё более простой и краткий синтаксис для создания функций, который часто лучше, чем синтаксис Function Expression.

Он называется «функции-стрелки» или «стрелочные функции» (arrow functions), т.к. выглядит следующим образом:

*let func = (arg1, arg2, ...argN) => expression*

Такой код создаёт функцию func с аргументами arg1..argN и вычисляет expression с правой стороны с их использованием, возвращая результат (Листинг 2.55).

**Листинг 2.55** Пример стрелочной функции

**let** *sum* = (a, b) => a + b;

*/\* Более короткая форма для:*

*let sum = function(a, b) {*

*return a + b;*

*};*

*\*/*

*alert*( *sum*(1, 2) ); *// 3*

То есть, (a, b) => a + b задаёт функцию с двумя аргументами a и b, которая при запуске вычисляет выражение справа a + b и возвращает его результат.

Если у нас только один аргумент, то круглые скобки вокруг параметров можно опустить, сделав запись ещё короче (Листинг 2.56).

**Листинг 2.56** Пример стрелочной функции без скобок

*// тоже что и*

*// let double = function(n) { return n \* 2 }*

**let** *double* = n => n \* 2;

*alert*( *double*(3) ); *// 6*

Если нет аргументов, указываются пустые круглые скобки (Листинг 2.57).

**Листинг 2.57** Пример стрелочной функции без входных параметров

**let** *sayHi* = () => *alert*(**"Hello!"**);

*sayHi*();

Поначалу функции-стрелки могут показаться необычными и трудно читаемыми, но это быстро пройдет, как только глаза привыкнут к этим конструкциям.

Они очень удобны для простых однострочных действий, когда лень писать много букв.

В примерах выше аргументы использовались слева от =>, а справа вычислялось выражение с их значениями.

Порой нам нужно что-то посложнее, например, выполнить несколько инструкций. Это также возможно, нужно лишь заключить инструкции в фигурные скобки. И использовать return внутри них, как в обычной функции (Листинг 2.58).

**Листинг 2.58** Пример многострочной стрелочной функции

**let** *sum* = (a, b) => { *// фигурная скобка, открывающая тело многострочной функции*

**let** result = a + b;

**return** result; *// при фигурных скобках для возврата значения нужно явно вызвать return*

};

*alert*( *sum*(1, 2) ); *// 3*

### Практическая работа №9

1. Замените код Function Expression стрелочной функцией:

**function** *ask*(question, yes, no) {

**if** (*confirm*(question)) yes()

**else** no();

}

*ask*(

**"Вы согласны?"**,

**function**() { *alert*(**"Вы согласились."**); },

**function**() { *alert*(**"Вы отменили выполнение."**); }

);

## Проверочная работа №1

Выполните предложенное ниже задание согласно своему варианту, составьте **отчет** с **блок-схемами** всех алгоритмов, **кодом** и **результатами работы**.

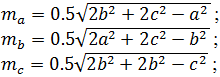
### Задание №1 (по вариантам)

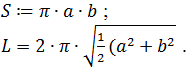
1. Вычислить площадь поверхности и объем усеченного конуса по следующим формулам:



1. Вычислить координаты центра тяжести трeх материальных точек с массами  и координатами по формулам:



1. Вычислить площадь треугольника со сторонами a, b, c по формуле Герона:  где p – полупериметр, вычисляемый по формуле (a+b+c)/2.
2. Вычислить координаты точки, делящей отрезок в отношении по формулам: где 
3. Вычислить медианы треугольника со сторонами a, b, c по формулам: 
4. Вычислить площадь круга и длину окружности по введенному значению радиуса.
5. Вычислить площадь S и периметр L эллипса по введенным значениям полуосей a и b:



1. Вычислить объем V и площадь боковой поверхности цилиндра S по введенным значениям радиуса основания R и высоты цилиндра H.



1. Вычислить объем V и площадь боковой поверхности конуса S по введенным значениям радиуса основания r, высоты h и образующей l:



1. Вычислить объем V и площадь поверхности S сферы по введенному значению радиуса r:



1. Дано целое четырехзначное число. Используя операции div и mod, найти сумму его цифр.
2. Дана сторона равностороннего треугольника. Найти площадь этого треугольника и радиусы вписанной и описанной окружностей.
3. Даны координаты трех вершин треугольника (x1, y1), (x2, y2), (x3, y3). Найти его периметр и площадь.
4. Дана длина окружности. Найти площадь круга, ограниченного этой окружностью.
5. Дана площадь круга. Найти длину окружности, ограничивающей этот круг.

### Задание №2 (по вариантам)

1. Составить программу для решения квадратного уравнения ax2 + bx + c = 0.
2. Определить максимальное четное число из двух введенных.
3. Определить, можно ли из отрезков с длинами х, y и z построить треугольник.
4. Ввести два числа а и b. Большее число заменить утроенным произведением, меньшее – полусуммой.
5. Если среди трех чисел a, b, c имеется хотя бы одно четное, то найти максимальное число, иначе – минимальное.
6. Определить, в каком квадранте находится точка с координатами x и y и вывести номер квадранта на экран.
7. Найти квадрат наибольшего из двух чисел а и b. Вывести на экран число 1, если наибольшим является число а, число 2 – если наибольшим числом является b.
8. Определить, попадает ли точка с координатами x и y в круг радиусом R. Если точка попадает в круг, вывести на экран единицу, в противном случае – ноль.
9. Написать алгоритм решения задачи, которая решает уравнение ax + b = 0 относительно x для любых чисел a и b, введенных с клавиатуры. Все числа считаются действительными.
10. Написать алгоритм решения задачи, которая определяет, лежит ли точка А(х,у) внутри некоторого кольца («внутри» понимается в строгом смысле, т.е. случай, когда точка А лежит на границе кольца, недопустим). Центр кольца находится в начале координат. Для кольца заданы внутренний и внешний радиусы r1, r2. Координаты x и у вводятся с клавиатуры.
11. . Даны две переменные целого типа: A и B. Если их значения не равны, то присвоить каждой переменной произведение этих значений, а если равны, то присвоить переменным нулевые значения.
12. Даны две переменные целого типа: A и B. Если их значения не равны, то присвоить каждой переменной минимальное из этих значений, а если равны, то присвоить переменным нулевые значения.
13. Даны целочисленные координаты точки на плоскости. Если точка не лежит на координатных осях, то вывести 0. Если точка совпадает с началом координат, то вывести 1. Если точка не совпадает с началом координат, но лежит на оси OX или OY, то вывести соответственно 2 или 3.
14. Даны вещественные координаты точки, не лежащей на координатных осях OX и OY. Вывести номер координатной четверти, в которой находится данная точка.
15. Дано целое число, лежащее в диапазоне от –999 до 999. Вывести строку – словесное описание данного числа вида "отрицательное двузначное число", "нулевое число", "положительное однозначное число" и т.д.

### Задание №3 (по вариантам)

1. Найти сумму чисел, кратных трем, в диапазоне от 0 до 50.
2. Найти сумму первых десяти чисел, кратных пяти.
3. Найти произведение четных чисел в диапазоне от 2 до 30.
4. Вводятся положительные числа. Прекратить ввод, когда сумма введенных чисел превысит 100.
5. Требуется найти сумму чисел, кратных 7, в диапазоне от 0 до 100. Вывести на экран сумму чисел и их количество.
6. Определить количество целых чисел, кратных 3 (от 3 и далее), дающих в сумме число, превышающее 200.
7. Вводятся 10 чисел. Вывести на экран суммы положительных и отрицательных чисел и их количество.
8. Вывести на экран значения функции у=sin(x) для 0≤x≤180 c шагом в 10.
9. Подсчитать площади десяти кругов с радиусами от 1 см с шагом 2 см и вывести значения площадей на экран.
10. Вводятся положительные числа. Прекратить ввод чисел, когда их сумма превысит 100. Результат вывести на экран.
11. Вводятся числа. Прекратить ввод чисел, когда сумма положительных чисел превысит 100. Результат вывести на экран.
12. Вывести на экран значения произведений чисел a и b. Числа а изменяются от 1 до 11 с шагом 1, b – от 1 до 3 с шагом 0,2.
13. Вывести на экран таблицу перевода километров в мили в диапазоне от 2 до 20 километров с шагом 2 км.
14. Вы положили в банк 1500 рублей. Определить, сколько денег будет на Вашем вкладе через 1 год, если каждый месяц вклад увеличивается на 0.76 % от суммы предыдущего месяца.
15. Решив заняться легкой атлетикой, Вы пробежали в первый день 2 км. Сколько километров Вы пробежите за 2 недели, если каждый день Вы увеличиваете дистанцию на 10 % от предыдущего дня?

### Критерии оценивания

3: Изучен теоретический материал 2 главы. Выполнены 2 задания. Составлен отчет о работе;

4: Изучен теоретический материал 2 главы. Выполнены задания (возможно с незначительными погрешностями). Составлен отчет о работе;

5: Изучен теоретический материал 2 главы. Выполнены задания. Составлен отчет о работе. Присутствует единый стиль оформления кода, комментарии.

# 

# 3. Основы работы с объектами

## Объекты

Как мы знаем, в JavaScript существует семь типов данных. Шесть из них называются «примитивными», так как содержат только одно значение (будь то строка, число или что-то другое).

Объекты же используются для хранения коллекций различных значений и более сложных сущностей. В JavaScript объекты используются очень часто, это одна из основ языка. Поэтому мы должны понять их, прежде чем углубляться куда-либо ещё.

Объект может быть создан с помощью фигурных скобок {…} с необязательным списком свойств. Свойство – это пара «ключ: значение», где ключ – это строка (также называемая «именем свойства»), а значение может быть чем угодно.

Мы можем представить объект в виде ящика с подписанными папками. Каждый элемент данных хранится в своей папке, на которой написан ключ. По ключу папку легко найти, удалить или добавить в неё что-либо (Рисунок 3.1).



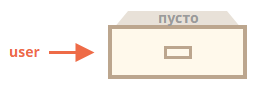
**Рисунок. 3.1.** Визуализация объекта

Пустой объект («пустой ящик») можно создать, используя один из двух вариантов синтаксиса (Листинг 3.1).

**Листинг 3.1** Создание объекта

**let *user*** = **new *Object***(); *// синтаксис "конструктор объекта"*

**let *user*** = {}; *// синтаксис "литерал объекта"*



**Рисунок. 3.2.** Создание объекта

Обычно используют вариант с фигурными скобками {...}. Такое объявление называют *литералом* объекта или *литеральной нотацией*.

### Литералы и свойства

При использовании литерального синтаксиса {...} мы сразу можем поместить в объект несколько свойств в виде пар «ключ: значение»:

**let *user*** = { *// объект*

**name**: **"John"**, *// под ключом "name" хранится значение "John"*

**age**: 30 *// под ключом "age" хранится значение 30*

};

Свойства объекта также иногда называют полями объекта.

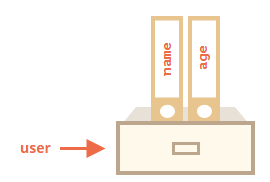
У каждого свойства есть ключ (также называемый «имя» или «идентификатор»). После имени свойства следует двоеточие ":", и затем указывается значение свойства. Если в объекте несколько свойств, то они перечисляются через запятую.

В объекте user сейчас находятся два свойства:

Первое свойство с именем "name" и значением "John".

Второе свойство с именем "age" и значением 30.

Можно сказать, что наш объект user – это ящик с двумя папками, подписанными «name» и «age» (Рисунок 3.3)



.

**Рисунок. 3.3.** Объект user

Мы можем в любой момент добавить в него новые папки, удалить папки или прочитать содержимое любой папки.

Для обращения к свойствам используется запись «через точку»:

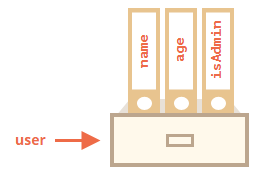
*// получаем свойства объекта:*

*alert*( user.**name** ); *// John*

*alert*( user.age ); *// 30*

Значение может быть любого типа. Давайте добавим свойство с логическим значением (Рисунок 3.4):

user.**isAdmin** = **true**;



**Рисунок. 3.4.** Объект user. Задание нового свойства

Для удаления свойства мы можем использовать оператор delete:

**delete** user.age;

Имя свойства может состоять из нескольких слов, но тогда оно должно быть заключено в кавычки (Рисунок 3.5):

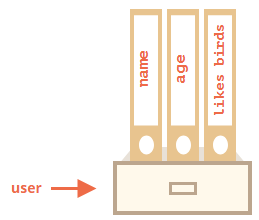
**let *user*** = {

**name**: **"John"**,

**age**: 30,

**"likes birds"**: **true** *// имя свойства из нескольких слов должно быть в кавычках*

};



**Рисунок. 3.5.** Объект user. Свойство состоящее из нескольких слов

### Квадратные скобки

Для свойств, имена которых состоят из нескольких слов, доступ к значению «через точку» не работает:

*// это вызовет синтаксическую ошибку*

user.likes **birds** = **true**

Так происходит, потому что точка требует, чтобы ключ был именован по правилам именования переменных. То есть не имел пробелов, не начинался с цифры и не содержал специальные символы, кроме $ и \_.

Для таких случаев существует альтернативный способ доступа к свойствам через квадратные скобки. Такой способ сработает с любым именем свойства (Листинг 3.2).

**Листинг 3.2** Обращение к свойствам с названиями состоящим из нескольких слов

**let *user*** = {};

*// присвоение значения свойству*

***user***[**"likes birds"**] = **true**;

*// получение значения свойства*

*alert*(***user***[**"likes birds"**]); *// true*

*// удаление свойства*

**delete *user***[**"likes birds"**];

Сейчас всё в порядке.

Обратите внимание, что строка в квадратных скобках закавычена (подойдёт любой тип кавычек).

Квадратные скобки также позволяют обратиться к свойству, имя которого может быть результатом выражения. Например, имя свойства может храниться в переменной (Листинг 3.3)

**Листинг 3.3** Обращение к свойствам через переменную

**let *key*** = **"likes birds"**;

*// то же самое, что и user["likes birds"] = true;*

user[***key***] = **true**;

Здесь переменная key может быть вычислена во время выполнения кода или зависеть от пользовательского ввода. После этого мы используем её для доступа к свойству. Это дает нам большую гибкость (Листинг 3.4).

**Листинг 3.4** Пример использования переменных для обращения к свойствам объекта

**let *user*** = {

**name**: **"John"**,

**age**: 30

};

**let *key*** = *prompt*(**"Что вы хотите узнать о пользователе?"**, **"name"**);

*// доступ к свойству через переменную*

*alert*( ***user***[***key***] ); *// John (если ввели "name")*

### Вычисляемые свойства

Мы можем использовать квадратные скобки в литеральной нотации для создания вычисляемого свойства (Листинг 3.5).

**Листинг 3.5** Вычисляемое свойство

**let *fruit*** = *prompt*(**"Какой фрукт купить?"**, **"apple"**);

**let *bag*** = {

[***fruit***]: 5, *// имя свойства будет взято из переменной fruit*

};

*alert*( ***bag***.**apple** ); *// 5, если fruit="apple"*

Смысл вычисляемого свойства прост: запись [fruit] означает, что имя свойства необходимо взять из переменной fruit.

И если посетитель введет слово "apple", то в объекте bag теперь будет лежать свойство {apple: 5}.

### Свойство из переменной

В реальном коде часто нам необходимо использовать существующие переменные как значения для свойств с тем же именем (Листинг 3.6).

**Листинг 3.6** Свойство из переменной с тем же именем

**function** *makeUser*(name, age) {

**return** {

**name**: name,

**age**: age

*// ...другие свойства*

};

}

**let *user*** = *makeUser*(**"John"**, 30);

*alert*(***user***.**name**); *// John*

В примере выше название свойств name и age совпадают с названиями переменных, которые мы подставляем в качестве значений этих свойств. Такой подход настолько распространён, что существуют специальные *короткие свойства* для упрощения этой записи (Листинг 3.7).

**Листинг 3.7** Свойство из переменной с тем же именем. Сокращенная запись

**function** *makeUser*(name, age) {

**return** {

name, *// то же самое, что и name: name*

age *// то же самое, что и age: age*

*// ...*

};

}

### Проверка существования свойства

Особенность объектов в том, что можно получить доступ к любому свойству. Даже если свойства не существует – ошибки не будет! При обращении к свойству, которого нет, возвращается undefined (Листинг 3.8).

**Листинг 3.8** Обращение к несуществующему свойству

**let *user*** = {};

*alert*( ***user***.noSuchProperty === **undefined** ); *// true означает "свойства нет"*

Также существует специальный оператор "in" для проверки существования свойства в объекте (Листинг 3.9).

**Листинг 3.9** Синтаксис оператора in

**let *user*** = { **name**: **"John"**, **age**: 30 };

*alert*( **"age" in *user*** ); *// true, user.age существует*

*alert*( **"blabla" in *user*** ); *// false, user.blabla не существует*

Обратите внимание, что слева от оператора in должно быть *имя свойства*. Обычно это строка в кавычках.

Если мы опускаем кавычки, это значит, что мы указываем переменную, в которой находится имя свойства.

### Цикл «for…in»

Для перебора всех свойств объекта используется цикл for..in. Этот цикл отличается от изученного ранее цикла for(;;).

Синтаксис:

**for** (**key in** object) {

*// тело цикла выполняется для каждого свойства объекта*

}

К примеру, давайте выведем все свойства объекта user (Листинг 3.10).

**Листинг 3.10** Перебор свойств объекта

**let *user*** = {

**name**: **"John"**,

**age**: 30,

**isAdmin**: **true**

};

**for** (**let *key* in *user***) {

*// ключи*

*alert*( ***key*** ); *// name, age, isAdmin*

*// значения ключей*

*alert*( ***user***[***key***] ); *// John, 30, true*

}

Обратите внимание, что все конструкции «for» позволяют нам объявлять переменную внутри цикла, как, например, let key здесь.

### Копирование по ссылке

Одним из фундаментальных отличий объектов от примитивных типов данных является то, что они хранятся и копируются «по ссылке».

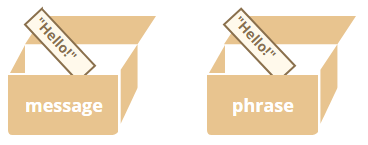
Примитивные типы: строки, числа, логические значения – присваиваются и копируются «по значению».

Например:

**let *message*** = **"Hello!"**;

**let *phrase*** = ***message***;

В результате мы имеем две независимые переменные, каждая из которых хранит строку "Hello!" (Рисунок 3.6).



**Рисунок. 3.6.** Объект user. Свойство состоящее из нескольких слов

Объекты ведут себя иначе.

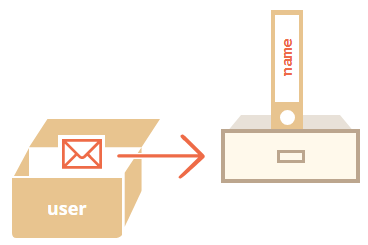
**Переменная хранит не сам объект, а его «адрес в памяти», другими словами «ссылку» на него.**

Проиллюстрируем это (Рисунок 3.7):

**let *user*** = {

**name**: **"John"**

};



**Рисунок. 3.7.** Организация данных в объекте

Сам объект хранится где-то в памяти. А в переменной user лежит «ссылка» на эту область памяти.

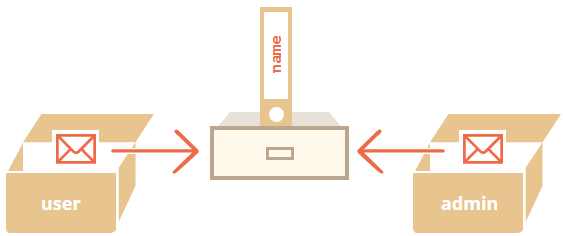
**Когда переменная объекта копируется – копируется ссылка, сам же объект не дублируется.**

Если мы представляем объект как ящик, то переменная – это ключ к нему. Копирование переменной дублирует ключ, но не сам ящик:

**let *user*** = { **name**: **"John"** };

**let *admin*** = ***user***; *// копируется ссылка*

Теперь у нас есть две переменные, каждая из которых содержит ссылку на один и тот же объект (Рисунок 3.8).



**Рисунок. 3.8.** Копирование объекта

Мы можем использовать любую из переменных для доступа к ящику и изменения его содержимого (Листинг 3.11).

**Листинг 3.11** Доступ к объекту

**let *user*** = { **name**: **'John'** };

**let *admin*** = ***user***;

***admin***.**name** = **'Pete'**; *// изменено по ссылке из переменной "admin"*

*alert*(***user***.**name**); *// 'Pete', изменения видны по ссылке из переменной "user"*

Приведённый выше пример демонстрирует, что объект только один. Как если бы у нас был один ящик с двумя ключами и мы использовали один из них (admin), чтобы войти в него и что-то изменить, а затем, открыв ящик другим ключом (user), мы бы увидели эти изменения.

### Сравнение объектов

Операторы равенства == и строгого равенства === для объектов работают одинаково.

**Два объекта равны только в том случае, если это один и тот же объект.**

### Клонирование и объединение объектов, Object.assign

Как мы узнали ранее, при копировании переменной объекта создаётся ещё одна ссылка на тот же самый объект.

Но что, если нам всё же нужно дублировать объект? Создать независимую копию, клон?

Это выполнимо, но немного сложно, так как в JavaScript нет встроенного метода для этого. На самом деле, такая нужда возникает редко. В большинстве случаев нам достаточно копирования по ссылке.

Но если мы действительно этого хотим, то нам нужно создавать новый объект и повторять структуру дублируемого объекта, перебирая его свойства и копируя их (Листинг 3.12).

**Листинг 3.12** Копирование объекта

**let *user*** = {

**name**: **"John"**,

**age**: 30

};

**let *clone*** = {}; *// новый пустой объект*

*// скопируем все свойства user в него*

**for** (**let *key* in *user***) {

***clone***[***key***] = ***user***[***key***];

}

*// теперь в переменной clone находится абсолютно независимый клон объекта.*

***clone***.**name** = **"Pete"**; *// изменим в нём данные*

*alert*( ***user***.**name** ); *// в оригинальном объекте значение свойства `name` осталось прежним – John.*

Кроме того, для этих целей мы можем использовать метод Object.assign.

Синтаксис:

*Object.assign(dest, [src1, src2, src3...])*

* Аргументы dest, и src1, ..., srcN (может быть столько, сколько нужно) являются объектами.
* Метод копирует свойства всех объектов src1, ..., srcN в объект dest. То есть, свойства всех перечисленных объектов, начиная со второго, копируются в первый объект. После копирования метод возвращает объект dest.

Например, объединим несколько объектов в один (Листинг 3.13).

**Листинг 3.13** Использование assign

**let *user*** = { **name**: **"John"** };

**let *permissions1*** = { **canView**: **true** };

**let *permissions2*** = { **canEdit**: **true** };

*// копируем все свойства из permissions1 и permissions2 в user*

***Object***.assign(***user***, ***permissions1***, ***permissions2***);

*// now user = { name: "John", canView: true, canEdit: true }*

Мы также можем использовать Object.assign для простого клонирования (Листинг 3.14).

**Листинг 3.14** Использование assign для простого клонирования

**let *user*** = {

**name**: **"John"**,

**age**: 30

};

**let *clone*** = ***Object***.assign({}, ***user***);

Все свойства объекта user будут скопированы в пустой объект, и ссылка на этот объект будет в переменной clone. На самом деле, такое клонирование работает так же, как и через цикл, но короче.

### Практическая работа №10

1. Напишите код, выполнив задание из каждого пункта отдельной строкой:
   1. Создайте пустой объект user.
   2. Добавьте свойство name со значением John.
   3. Добавьте свойство surname со значением Smith.
   4. Измените значение свойства name на Pete.
   5. Удалите свойство name из объекта.
2. Напишите функцию isEmpty(obj), которая возвращает true, если у объекта нет свойств, иначе false.

Должно работать так:

**let *schedule*** = {};

*alert*( isEmpty(***schedule***) ); *// true*

***schedule***[**"8:30"**] = **"get up"**;

*alert*( isEmpty(***schedule***) ); *// false*

## Методы объекта, "this"

Объекты обычно создаются, чтобы представлять сущности реального мира, будь то пользователи, заказы и так далее.

И так же, как и в реальном мире, пользователь может совершать действия: выбирать что-то из корзины покупок, авторизовываться, выходить из системы, оплачивать и т.п.

Такие действия в JavaScript представлены свойствами-функциями объекта.

### Примеры методов

Для начала давайте научим пользователя user здороваться (Листинг 3.15).

**Листинг 3.15** Пример метода

**let *user*** = {

**name**: **"Джон"**,

**age**: 30

};

***user***.sayHi = **function**() {

*alert*(**"Привет!"**);

};

***user***.sayHi(); *// Привет!*

Здесь мы просто использовали Function Expression (функциональное выражение), чтобы создать функцию для приветствия, и присвоили её свойству user.sayHi нашего объекта.

Затем мы вызвали её. Теперь пользователь может говорить!

Функцию, которая является свойством объекта, называют *методом* этого объекта.

Итак, мы получили метод sayHi объекта user.

### Сокращённая запись метода

Существует более короткий синтаксис для методов в литерале объекта (Листинг 3.16).

**Листинг 3.16** Сокращенная запись метода

*// эти объекты делают одно и то же (одинаковые методы)*

**user** = {

sayHi: **function**() {

*alert*(**"Привет"**);

}

};

*// сокращённая запись выглядит лучше, не так ли?*

**user** = {

sayHi() { *// то же самое, что и "sayHi: function()"*

*alert*(**"Привет"**);

}

};

### Ключевое слово «this» в методах

Как правило, методу объекта необходим доступ к информации, которая хранится в объекте, чтобы выполнить с ней какие-либо действия (в соответствии с назначением метода).

Например, коду внутри user.sayHi() может понадобиться имя пользователя, которое хранится в объекте user.

**Для доступа к информации внутри объекта метод может использовать ключевое слово this.**

Значение this – это объект «перед точкой», который использовался для вызова метода (Листинг 3.17).

**Листинг 3.17** Использование this

**let *user*** = {

**name**: **"Джон"**,

**age**: 30,

sayHi() {

*// this - это "текущий объект"*

*alert*(**this**.**name**);

}

};

***user***.sayHi(); *// Джон*

Здесь во время выполнения кода user.sayHi() значением this будет являться user (ссылка на объект user).

Технически также возможно получить доступ к объекту без ключевого слова this, ссылаясь на него через внешнюю переменную (в которой хранится ссылка на этот объект) (Листинг 3.18)

**Листинг 3.18** Код без использование this

**let *user*** = {

**name**: **"Джон"**,

**age**: 30,

sayHi() {

*alert*(***user***.**name**); *// используем переменную "user" вместо ключевого слова "this"*

}

};

…Но такой код будет ненадежным. Если мы решим скопировать ссылку на объект user в другую переменную, например, admin = user, и перепишем переменную user чем-то другим, тогда будет осуществлен доступ к неправильному объекту при вызове метода из admin.

### «this» не является фиксированным

В JavaScript ключевое слово «this» ведёт себя иначе, чем в большинстве других языков программирования. Оно может использоваться в любой функции (Листинг 3.19).

**Листинг 3.19** this в функции

**function** *sayHi*() {

*alert*( **this**.**name** );

}

Значение this вычисляется во время выполнения кода и зависит от контекста.

Например, здесь одна и та же функция назначена двум разным объектам и имеет различное значение «this» при вызовах (Листинг 3.20).

**Листинг 3.20** this в зависимости от контекста

**let *user*** = { **name**: **"Джон"** };

**let *admin*** = { **name**: **"Админ"** };

**function** *sayHi*() {

*alert*( **this**.**name** );

}

*// используем одну и ту же функцию в двух объектах*

user.f = sayHi;

admin.f = sayHi;

*// вызовы функции, приведённые ниже, имеют разное значение this*

*// "this" внутри функции является ссылкой на объект, который указан "перед точкой"*

user.f(); *// Джон (this == user)*

admin.f(); *// Админ (this == admin)*

admin[**'f'**](); *// Админ (не важен способ доступа к методу - через точку или квадратные скобки)*

Правило простое: при вызове obj.f() значение this внутри f равно obj. Так что, в приведённом примере это user или admin.

### У стрелочных функций нет «this»

Стрелочные функции особенные: у них нет своего «собственного» this. Если мы используем this внутри стрелочной функции, то его значение берётся из внешней «нормальной» функции.

Например, здесь arrow() использует значение this из внешнего метода user.sayHi() (Листинг 3.21).

**Листинг 3.21** this в стрелочной функции

**let *user*** = {

**firstName**: **"Илья"**,

sayHi() {

**let** *arrow* = () => *alert*(**this**.**firstName**);

*arrow*();

}

};

***user***.sayHi(); *// Илья*

Это является особенностью стрелочных функций. Они полезны, когда мы на самом деле не хотим иметь отдельное значение this, а хотим брать его из внешнего контекста.

### Практическая работа №11

1. Создайте объект calculator (калькулятор) с тремя методами:
   1. read() (читать) запрашивает два значения и сохраняет их как свойства объекта.
   2. sum() (суммировать) возвращает сумму сохранённых значений.
   3. mul() (умножить) перемножает сохраненные значения и возвращает результат.

**let *calculator*** = {

*// ... ваш код ...*

};

***calculator***.read();

*alert*( ***calculator***.sum() );

*alert*( ***calculator***.mul() );

1. Это ladder (лестница) – объект, который позволяет подниматься вверх и спускаться:

**let *ladder*** = {

**step**: 0,

up() {

**this**.**step**++;

},

down() {

**this**.**step**--;

},

showStep: **function**() { *// показывает текущую ступеньку*

*alert*( **this**.**step** );

}

};

Теперь, если нам нужно сделать несколько последовательных вызовов, мы можем выполнить это так:

ladder.up();

ladder.up();

ladder.down();

ladder.showStep(); *// 1*

Измените код методов up, down и showStep таким образом, чтобы их вызов можно было сделать по цепочке, например так:

ladder.up().up().down().showStep(); *// 1*

Такой подход широко используется в библиотеках JavaScript.

## Преобразование объектов в примитивы

Что произойдёт, если сложить два объекта obj1 + obj2, вычесть один из другого obj1 - obj2 или вывести их на экран, воспользовавшись alert(obj)?

В этом случае объекты сначала автоматически преобразуются в примитивы, а затем выполняется операция.

В главе Преобразование типов мы видели правила для числовых, строковых и логических преобразований. Но обделили вниманием объекты. Теперь, поскольку мы уже знаем о методах объектов и символах, можно исправить это упущение.

1. Все объекты в логическом контексте являются true. Существуют лишь их численные и строковые преобразования.
2. Численные преобразования происходят, когда мы вычитаем объекты или выполняем математические операции. Например, объекты Date (мы рассмотрим их в статье Дата и время) могут вычитаться, и результатом date1 - date2 будет временной отрезок между двумя датами.
3. Что касается строковых преобразований – они обычно происходят, когда мы выводим объект alert(obj), а также в других случаях, когда объект используется как строка.

### Преобразование к примитивам

Мы можем тонко настраивать строковые и численные преобразования, используя специальные методы объекта.

Существуют три варианта преобразований («три хинта»), описанные в спецификации:

**"string"**

Для преобразования объекта к строке, когда операция ожидает получить строку, например alert (Листинг 3.21)

**Листинг 3.21** Преобразование объекта к строке

*// вывод*

*alert*(obj);

*// используем объект в качестве имени свойства*

anotherObj[obj] = 123;

**"number"**

Для преобразования объекта к числу, в случае математических операций (Листинг 3.22)

**Листинг 3.22** Преобразование объекта к числу

*// явное преобразование*

**let *num*** = ***Number***(obj);

*// математическое (исключая бинарный оператор "+")*

**let *n*** = +obj; *// унарный плюс*

**let *delta*** = date1 - date2;

*// сравнения больше/меньше*

**let *greater*** = user1 > user2;

**"default"**

Происходит редко, когда оператор «не уверен», какой тип ожидать.

Например, бинарный плюс + может работать с обоими типами: строками (объединять их) и числами (складывать). Таким образом, и те, и другие будут вычисляться. Или когда происходит сравнение объектов с помощью нестрогого равенства == со строкой, числом или символом, и неясно, какое преобразование должно быть выполнено (Листинг 3.23).

**Листинг 3.23** Преобразование “default”

*// бинарный плюс*

**let *total*** = car1 + car2;

*// obj == string/number/symbol*

**if** (user == 1) { ... };

Оператор больше/меньше <> также может работать как со строками, так и с числами. Однако, по историческим причинам он использует хинт «number», а не «default».

На практике все встроенные объекты, исключая Date (мы познакомимся с ним чуть позже), реализуют "default" преобразования тем же способом, что и "number". И нам следует поступать также.

Обратите внимание, что существуют лишь три варианта хинтов. Всё настолько просто. Не существует хинта со значением «boolean» (все объекты являются true в логическом контексте) или каких-либо ещё. И если мы считаем "default" и "number" одинаковыми, как большинство встроенных объектов, то остаются всего два варианта преобразований.

**В процессе преобразования движок JavaScript пытается найти и вызвать три следующих метода объекта:**

1. Вызывает obj[Symbol.toPrimitive](hint) – метод с символьным ключом Symbol.toPrimitive (системный символ), если такой метод существует, и передаёт ему хинт.
2. Иначе, если хинт равен "string"
   1. пытается вызвать obj.toString(), а если его нет, то obj.valueOf(), если он существует.
3. В случае, если хинт равен "number" или "default"
   1. пытается вызвать obj.valueOf(), а если его нет, то obj.toString(), если он существует.

### Symbol.toPrimitive

Начнём с универсального подхода – символа Symbol.toPrimitive: метод с таким названием (если есть) используется для всех преобразований:

obj[***Symbol***.**toPrimitive**] = **function**(hint) {

*// должен вернуть примитивное значение*

*// hint равно чему-то одному из: "string", "number" или "default"*

};

Для примера используем его в реализации объекта user (Листинг 3.24):

**Листинг 3.24** Использование Symbol.toPrimitive

**let *user*** = {

**name**: **"John"**,

**money**: 1000,

[***Symbol***.**toPrimitive**](hint) {

*alert*(**`hint:** ${hint}**`**);

**return** hint == **"string"** ? **`{name: "**${**this**.**name**}**"}`** : **this**.**money**;

}

};

*// демонстрация результатов преобразований:*

*alert*(***user***); *// hint: string -> {name: "John"}*

*alert*(+***user***); *// hint: number -> 1000*

*alert*(***user*** + 500); *// hint: default -> 1500*

Как мы видим из кода, user преобразовывается либо в информативную читаемую строку, либо в денежный счёт в зависимости от значения хинта. Единственный метод user[Symbol.toPrimitive] смог обработать все случаи преобразований.

### Методы toString/valueOf

Методы toString и valueOf берут своё начало с древних времён. Они не символы, так как в то время символов ещё не существовало, а просто обычные методы объектов со строковыми именами. Они предоставляют «устаревший» способ реализации преобразований объектов.

Если нет метода Symbol.toPrimitive, движок JavaScript пытается найти эти методы и вызвать их следующим образом:

* toString -> valueOf для хинта со значением «string».
* valueOf -> toString – в ином случае.

Для примера, используем их в реализации всё того же объекта user. Воспроизведём его поведение комбинацией методов toString и valueOf (Листинг 3.25)

**Листинг 3.25** Использование toString и valueOf

**let *user*** = {

**name**: **"John"**,

**money**: 1000,

*// для хинта равного "string"*

toString() {

**return `{name: "**${**this**.**name**}**"}`**;

},

*// для хинта равного "number" или "default"*

valueOf() {

**return this**.**money**;

}

};

*alert*(***user***); *// toString -> {name: "John"}*

*alert*(+***user***); *// valueOf -> 1000*

*alert*(***user*** + 500); *// valueOf -> 1500*

Как видим, получилось то же поведение, что и в предыдущем примере с Symbol.toPrimitive.

Довольно часто мы хотим описать одно «универсальное» преобразование объекта к примитиву для всех ситуаций. Для этого достаточно создать один toString (Листинг 3.26)

**Листинг 3.26** Использование одного toString

**let *user*** = {

**name**: **"John"**,

toString() {

**return this**.**name**;

}

};

*alert*(***user***); *// toString -> John*

*alert*(***user*** + 500); *// toString -> John500*

В отсутствие Symbol.toPrimitive и valueOf, toString обработает все случаи преобразований к примитивам.

### Возвращаемые типы

Важно понимать, что все описанные методы для преобразований объектов не обязаны возвращать именно требуемый «хинтом» тип примитива.

Нет обязательного требования, чтобы toString() возвращал именно строку, или чтобы метод Symbol.toPrimitive возвращал именно число для хинта «number».

**Единственное обязательное требование: методы должны возвращать примитив, а не объект.**

## Конструкторы, создание объектов через "new"

Обычный синтаксис {...} позволяет создать только один объект. Но зачастую нам нужно создать множество однотипных объектов, таких как пользователи, элементы меню и т.д.

Это можно сделать при помощи функции-конструктора и оператора "new".

### Функция-конструктор

Функции-конструкторы являются обычными функциями. Но есть два соглашения:

1. Имя функции-конструктора должно начинаться с большой буквы.
2. Функция-конструктор должна вызываться при помощи оператора "new".

Например (Листинг 3.27):

**Листинг 3.27** Использование одного toString

**function** *User*(name) {

**this**.**name** = name;

**this**.**isAdmin** = **false**;

}

**let *user*** = **new** *User*(**"Вася"**);

*alert*(***user***.**name**); *// Вася*

*alert*(***user***.**isAdmin**); *// false*

Когда функция вызывается как new User(...), происходит следующее:

1. Создаётся новый пустой объект, и он присваивается this.
2. Выполняется код функции. Обычно он модифицирует this, добавляет туда новые свойства.
3. Возвращается значение this.

То есть, результат вызова new User("Вася") – это тот же объект, что и:

**let *user*** = {

**name**: **"Вася"**,

**isAdmin**: **false**

};

Теперь, когда нам необходимо будет создать других пользователей, мы можем использовать new User("Маша"), new User("Даша") и т.д. Данная конструкция гораздо удобнее и читабельнее, чем каждый раз создавать литерал объекта. Это и является основной целью конструкторов – удобное повторное создание однотипных объектов.

Ещё раз заметим: технически любая функция может быть использована как конструктор. То есть, каждая функция может быть вызвана при помощи оператора new, и выполнится алгоритм, указанный выше в примере. Заглавная буква в названии функции является всеобщим соглашением по именованию, она как бы подсказывает разработчику, что данная функция является функцией-конструктором, и её нужно вызывать через new.

### Создание методов в конструкторе

Использование конструкторов для создания объектов даёт большую гибкость. Можно передавать конструктору параметры, определяющие, как создавать объект, и что в него записывать.

В this мы можем добавлять не только свойства, но и методы.

Например, в примере ниже, new User(name) создаёт объект с данным именем name и методом sayHi (Листинг 3.28):

**Листинг 3.28** Создание методов в конструкторе

**function** *User*(name) {

**this**.**name** = name;

**this**.sayHi = **function**() {

*alert*( **"Меня зовут: "** + **this**.**name** );

};

}

**let *vasya*** = **new** *User*(**"Вася"**);

***vasya***.sayHi(); *// Меня зовут: Вася*

*/\**

*vasya = {*

*name: "Вася",*

*sayHi: function() { ... }*

*}*

*\*/*

Для создания сложных объектов есть и более «продвинутый» синтаксис – классы, которые мы разберём позже.

### Практическая работа №12

1. Создайте функцию-конструктор Calculator, который создаёт объекты с тремя методами:
   1. read() запрашивает два значения при помощи prompt и сохраняет их значение в свойствах объекта.
   2. sum() возвращает сумму введённых свойств.
   3. mul() возвращает произведение введённых свойств.

Например:

**let *calculator*** = **new** Calculator();

***calculator***.read();

*alert*( **"Sum="** + ***calculator***.sum() );

*alert*( **"Mul="** + ***calculator***.mul() );

1. Напишите функцию-конструктор Accumulator(startingValue).

Объект, который она создаёт, должен уметь следующее:

* Хранить «текущее значение» в свойстве value. Начальное значение устанавливается в аргументе конструктора startingValue.
* Метод read() использует prompt для получения числа и прибавляет его к свойству value.

Таким образом, свойство value является текущей суммой всего, что ввёл пользователь при вызовах метода read(), с учётом начального значения startingValue.

Ниже вы можете посмотреть работу кода:

**let *accumulator*** = **new** Accumulator(1); *// начальное значение 1*

***accumulator***.read(); *// прибавит ввод prompt к текущему значению*

***accumulator***.read(); *// прибавит ввод prompt к текущему значению*

*alert*(***accumulator***.**value**); *// выведет сумму этих значений*

# 

# 4. Основы работы с массивами

## Массивы

Объекты позволяют хранить данные со строковыми ключами. Это замечательно.

Но довольно часто мы понимаем, что нам необходима упорядоченная коллекция данных, в которой присутствуют 1-й, 2-й, 3-й элементы и т.д. Например, она понадобится нам для хранения списка чего-либо: пользователей, товаров, элементов HTML и т.д.

В этом случае использовать объект неудобно, так как он не предоставляет методов управления порядком элементов. Мы не можем вставить новое свойство «между» уже существующими. Объекты просто не предназначены для этих целей.

Для хранения упорядоченных коллекций существует особая структура данных, которая называется массив, Array.

### Объявление

Существует два варианта синтаксиса для создания пустого массива:

**let *arr*** = **new *Array***();

**let *arr*** = [];

Практически всегда используется второй вариант синтаксиса. В скобках мы можем указать начальные значения элементов:

**let *fruits*** = [**"Яблоко"**, **"Апельсин"**, **"Слива"**];

Элементы массива нумеруются, начиная с нуля.

Мы можем получить элемент, указав его номер в квадратных скобках (Листинг 4.1):

**Листинг 4.1** Обращение к элементам в массиве

**let *fruits*** = [**"Яблоко"**, **"Апельсин"**, **"Слива"**];

*alert*( ***fruits***[0] ); *// Яблоко*

*alert*( ***fruits***[1] ); *// Апельсин*

*alert*( ***fruits***[2] ); *// Слива*

Мы можем заменить элемент:

fruits[2] = **'Груша'**; *// теперь ["Яблоко", "Апельсин", "Груша"]*

…Или добавить новый к существующему массиву:

fruits[3] = **'Лимон'**; *// теперь ["Яблоко", "Апельсин", "Груша", "Лимон"]*

Общее число элементов массива содержится в его свойстве length (Листинг 4.2):

**Листинг 4.2** Получение количества элементов в массиве

**let *fruits*** = [**"Яблоко"**, **"Апельсин"**, **"Слива"**];

*alert*( ***fruits***.**length** ); *// 3*

Вывести массив целиком можно при помощи alert (Листинг 4.3).

**Листинг 4.3** Вывод элементов массива

**let *fruits*** = [**"Яблоко"**, **"Апельсин"**, **"Слива"**];

*alert*( ***fruits*** ); *// Яблоко, Апельсин, Слива*

В массиве могут храниться элементы любого типа (Листинг 4.4).

**Листинг 4.4** Хранение в массиве элементов разного типа

*// разные типы значений*

**let *arr*** = [ **'Яблоко'**, { **name**: **'Джон'** }, **true**, **function**() { *alert*(**'привет'**); } ];

*// получить элемент с индексом 1 (объект) и затем показать его свойство*

*alert*( ***arr***[1].**name** ); *// Джон*

*// получить элемент с индексом 3 (функция) и выполнить её*

***arr***[3](); *// привет*

### Методы pop/push, shift/unshift

*Очередь* – один из самых распространенных вариантов применения массива. В области компьютерных наук так называется упорядоченная коллекция элементов, поддерживающая два вида операций:

* push добавляет элемент в конец.
* shift удаляет элемент в начале, сдвигая очередь, так что второй элемент становится первым.



**Рисунок 4.1.** Визуализация очереди

Массивы поддерживают обе операции.

На практике необходимость в этом возникает очень часто. Например, очередь сообщений, которые надо показать на экране.

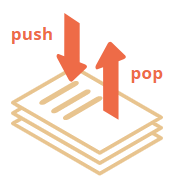
Существует и другой вариант применения для массивов – структура данных, называемая *стек*.

Она поддерживает два вида операций:

* push добавляет элемент в конец.
* pop удаляет последний элемент.

Таким образом, новые элементы всегда добавляются или удаляются из «конца».

Примером стека обычно служит колода карт: новые карты кладутся наверх и берутся тоже сверху (Рисунок 4.2)



**Рисунок 4.2.** Визуализация стека

Массивы в JavaScript могут работать и как очередь, и как стек. Мы можем добавлять/удалять элементы как в начало, так и в конец массива.

В компьютерных науках структура данных, делающая это возможным, называется *двусторонняя очередь*.

**Методы, работающие с концом массива:**

**pop**

Удаляет последний элемент из массива и возвращает его (Листинг 4.5).

**Листинг 4.5** Метод pop

**let *fruits*** = [**"Яблоко"**, **"Апельсин"**, **"Груша"**];

*alert*( ***fruits***.pop() ); *// удаляем "Груша" и выводим его*

*alert*( ***fruits*** ); *// Яблоко, Апельсин*

**push**

Добавляет элемент в конец массива (Листинг 4.6).

**Листинг 4.6** Метод push

***let fruits*** *= [****"Яблоко"****,* ***"Апельсин"****];*

***fruits****.push(****"Груша"****);*

*alert(* ***fruits*** *); // Яблоко, Апельсин, Груша*

**Методы, работающие с началом массива:**

**shift**

Удаляет из массива первый элемент и возвращает его (Листинг 4.7).

**Листинг 4.7** Метод shift

**let *fruits*** = [**"Яблоко"**, **"Апельсин"**, **"Груша"**];

*alert*( ***fruits***.shift() ); *// удаляем Яблоко и выводим его*

*alert*( ***fruits*** ); *// Апельсин, Груша*

**unshift**

Добавляет элемент в начало массива (Листинг 4.8).

**Листинг 4.8** Метод unshift

**let *fruits*** = [**"Апельсин"**, **"Груша"**];

***fruits***.unshift(**'Яблоко'**);

*alert*( ***fruits*** ); *// Яблоко, Апельсин, Груша*

### Перебор элементов

Одним из самых старых способов перебора элементов массива является цикл for по цифровым индексам (Листинг 4.9).

**Листинг 4.9** Перебор массива с помощью цикла for

**let *arr*** = [**"Яблоко"**, **"Апельсин"**, **"Груша"**];

**for** (**let *i*** = 0; ***i*** < ***arr***.**length**; ***i***++) {

*alert*( ***arr***[***i***] );

}

Но для массивов возможен и другой вариант цикла, for..of (Листинг 4.10).

**Листинг 4.10** Перебор массива с помощью цикла for … of

**let *fruits*** = [**"Яблоко"**, **"Апельсин"**, **"Слива"**];

*// проходит по значениям*

**for** (**let *fruit* of *fruits***) {

*alert*( ***fruit*** );

}

Цикл for..of не предоставляет доступа к номеру текущего элемента, только к его значению, но в большинстве случаев этого достаточно. А также это короче.

### Многомерные массивы

Массивы могут содержать элементы, которые тоже являются массивами. Это можно использовать для создания многомерных массивов, например, для хранения матриц (Листинг 4.11).

**Листинг 4.11** Многомерные массивы

**let *matrix*** = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

];

*alert*( ***matrix***[1][1] ); *// 5, центральный элемент*

### toString

Массивы по-своему реализуют метод toString, который возвращает список элементов, разделенных запятыми (Листинг 4.12).

**Листинг 4.12** Реализация метода toString в массивах

**let *arr*** = [1, 2, 3];

*alert*( ***arr*** ); *// 1,2,3*

*alert*( ***String***(***arr***) === **'1,2,3'** ); *// true*

### Практическая работа №13

1. Напишите функцию sumInput(), которая:
   1. Просит пользователя ввести значения, используя prompt и сохраняет их в массив.
   2. Заканчивает запрашивать значения, когда пользователь введёт не числовое значение, пустую строку или нажмёт «Отмена».
   3. Подсчитывает и возвращает сумму элементов массива.

P.S. Ноль 0 – считается числом, не останавливайте ввод значений при вводе «0».

1. На входе массив чисел, например: arr = [1, -2, 3, 4, -9, 6].

Задача: найти непрерывный подмассив в arr, сумма элементов в котором максимальна.

Функция getMaxSubSum(arr) должна возвращать эту сумму.

Например:

getMaxSubSum([-1, 2, 3, -9]) = 5 (сумма выделенных)

getMaxSubSum([2, -1, 2, 3, -9]) = 6

getMaxSubSum([-1, 2, 3, -9, 11]) = 11

getMaxSubSum([-2, -1, 1, 2]) = 3

getMaxSubSum([100, -9, 2, -3, 5]) = 100

getMaxSubSum([1, 2, 3]) = 6 (берём все)

Если все элементы отрицательные – ничего не берём(подмассив пустой) и сумма равна «0».

## Методы массивов

Массивы предоставляют множество методов. Чтобы было проще, в этой главе они разбиты на группы.

### Добавление/удаление элементов

#### splice

Метод arr.splice(str) – это универсальный «швейцарский нож» для работы с массивами. Умеет всё: добавлять, удалять и заменять элементы.

Его синтаксис:

*arr.splice(index[, deleteCount, elem1, ..., elemN])*

Он начинает с позиции index, удаляет deleteCount элементов и вставляет elem1, ..., elemN на их место. Возвращает массив из удаленных элементов.

Этот метод проще всего понять, рассмотрев примеры.

Начнем с удаления (Листинг 4.13).

**Листинг 4.13** Удаление элементов с помощью метода splice

**let *arr*** = [**"Я"**, **"изучаю"**, **"JavaScript"**];

***arr***.splice(1, 1); *// начиная с позиции 1, удалить 1 элемент*

*alert*( ***arr*** ); *// осталось ["Я", "JavaScript"]*

Легко, правда? Начиная с позиции 1, он убрал 1 элемент.

В следующем примере мы удалим 3 элемента и заменим их двумя другими (Листинг 4.14).

**Листинг 4.14** Замена элементов с помощью метода splice

**let *arr*** = [**"Я"**, **"изучаю"**, **"JavaScript"**, **"прямо"**, **"сейчас"**];

*// удалить 3 первых элемента и заменить их другими*

***arr***.splice(0, 3, **"Давай"**, **"танцевать"**);

*alert*( ***arr*** ) *// теперь ["Давай", "танцевать", "прямо", "сейчас"]*

Здесь видно, что splice возвращает массив из удалённых элементов (Листинг 4.15).

**Листинг 4.15** Возвращение методом splice удаленных элементов

**let *arr*** = [**"Я"**, **"изучаю"**, **"JavaScript"**, **"прямо"**, **"сейчас"**];

*// удалить 2 первых элемента*

**let *removed*** = ***arr***.splice(0, 2);

*alert*( ***removed*** ); *// "Я", "изучаю" <-- массив из удалённых элементов*

Метод splice также может вставлять элементы без удаления, для этого достаточно установить deleteCount в 0 (Листинг 4.16).

**Листинг 4.16** Вставка элементов методом splice

**let *arr*** = [**"Я"**, **"изучаю"**, **"JavaScript"**];

*// с позиции 2*

*// удалить 0 элементов*

*// вставить "сложный", "язык"*

***arr***.splice(2, 0, **"сложный"**, **"язык"**);

*alert*( ***arr*** ); *// "Я", "изучаю", "сложный", "язык", "JavaScript"*

#### slice

Метод arr.slice намного проще, чем похожий на него arr.splice.

Его синтаксис:

*arr.slice([start], [end])*

Он возвращает новый массив, в который копирует элементы, начиная с индекса start и до end (не включая end). Оба индекса start и end могут быть отрицательными. В таком случае отсчет будет осуществляться с конца массива.

Можно вызвать slice и вообще без аргументов: arr.slice() создает копию массива arr. Это часто используют, чтобы создать копию массива для дальнейших преобразований, которые не должны менять исходный массив.

#### concat

Метод arr.concat создаёт новый массив, в который копирует данные из других массивов и дополнительные значения.

Его синтаксис:

*arr.concat(arg1, arg2...)*

Он принимает любое количество аргументов, которые могут быть как массивами, так и простыми значениями.

В результате мы получаем новый массив, включающий в себя элементы из arr, а также arg1, arg2 и так далее…

Если аргумент argN – массив, то все его элементы копируются. Иначе скопируется сам аргумент (Листинг 4.17).

**Листинг 4.17** Метод concat

**let *arr*** = [1, 2];

*// создать массив из: arr и [3,4]*

*alert*( ***arr***.concat([3, 4]) ); *// 1,2,3,4*

*// создать массив из: arr и [3,4] и [5,6]*

*alert*( ***arr***.concat([3, 4], [5, 6]) ); *// 1,2,3,4,5,6*

*// создать массив из: arr и [3,4], потом добавить значения 5 и 6*

*alert*( ***arr***.concat([3, 4], 5, 6) ); *// 1,2,3,4,5,6*

### Перебор: forEach

Метод arr.forEach позволяет запускать функцию для каждого элемента массива.

Его синтаксис:

*arr.forEach(function(item, index, array) {*

*// ... делать что-то с item*

*});*

Например, этот код выведет на экран каждый элемент массива (Листинг 4.18).

**Листинг 4.18** Метод foreach

*// Вызов alert для каждого элемента*

[**"Bilbo"**, **"Gandalf"**, **"Nazgul"**].forEach(*alert*);

### Поиск в массиве

Далее рассмотрим методы, которые помогут найти что-нибудь в массиве.

#### indexOf/lastIndexOf и includes

Методы arr.indexOf, arr.lastIndexOf и arr.includes имеют одинаковый синтаксис и делают по сути то же самое, что и их строковые аналоги, но работают с элементами вместо символов:

* arr.indexOf(item, from) ищет item, начиная с индекса from, и возвращает индекс, на котором был найден искомый элемент, в противном случае -1.
* arr.lastIndexOf(item, from) – то же самое, но ищет справа налево.
* arr.includes(item, from) – ищет item, начиная с индекса from, и возвращает true, если поиск успешен.

Например (Листинг 4.19):

**Листинг 4.19** Поиск элементов в массиве

**let *arr*** = [1, 0, **false**];

*alert*( ***arr***.indexOf(0) ); *// 1*

*alert*( ***arr***.indexOf(**false**) ); *// 2*

*alert*( ***arr***.indexOf(**null**) ); *// -1*

*alert*( ***arr***.includes(1) ); *// true*

Обратите внимание, что методы используют строгое сравнение ===. Таким образом, если мы ищем false, он находит именно false, а не ноль.

Если мы хотим проверить наличие элемента, и нет необходимости знать его точный индекс, тогда предпочтительным является arr.includes.

Кроме того, очень незначительным отличием includes является то, что он правильно обрабатывает NaN в отличие от indexOf/lastIndexOf.

#### find и findIndex

Представьте, что у нас есть массив объектов. Как нам найти объект с определённым условием?

Здесь пригодится метод arr.find.

Его синтаксис таков:

*let result = arr.find(function(item, index, array) {*

*// если true - возвращается текущий элемент и перебор прерывается*

*// если все итерации оказались ложными, возвращается undefined*

*});*

Функция вызывается по очереди для каждого элемента массива:

* item – очередной элемент.
* index – его индекс.
* array – сам массив.

Если функция возвращает true, поиск прерывается и возвращается item. Если ничего не найдено, возвращается undefined.

Например, у нас есть массив пользователей, каждый из которых имеет поля id и name. Попробуем найти того, кто с id == 1 (Листинг 4.20):

**Листинг 4.20** Использование метода find

**let *users*** = [

{**id**: 1, **name**: **"Вася"**},

{**id**: 2, **name**: **"Петя"**},

{**id**: 3, **name**: **"Маша"**}

];

**let *user*** = ***users***.find(item => item.**id** == 1);

*alert*(***user***.**name**); *// Вася*

В реальной жизни массивы объектов – обычное дело, поэтому метод find крайне полезен.

Обратите внимание, что в данном примере мы передаём find функцию item => item.id == 1, с одним аргументом. Это типично, дополнительные аргументы этой функции используются редко.

Метод arr.findIndex – по сути, то же самое, но возвращает индекс, на котором был найден элемент, а не сам элемент, и -1, если ничего не найдено.

#### filter

Метод find ищет один (первый попавшийся) элемент, на котором функция-колбэк вернёт true.

На тот случай, если найденных элементов может быть много, предусмотрен метод arr.filter(fn).

Синтаксис этого метода схож с find, но filter возвращает массив из всех подходящих элементов:

*let results = arr.filter(function(item, index, array) {*

*// если true - элемент добавляется к результату, и перебор продолжается*

*// возвращается пустой массив в случае, если ничего не найдено*

*});*

Пример (Листинг 4.21)

**Листинг 4.21** Использование метода filter

**let *users*** = [

{**id**: 1, **name**: **"Вася"**},

{**id**: 2, **name**: **"Петя"**},

{**id**: 3, **name**: **"Маша"**}

];

*// возвращает массив, состоящий из двух первых пользователей*

**let *someUsers*** = ***users***.filter(item => item.**id** < 3);

*alert*(***someUsers***.**length**); *// 2*

### Преобразование массива

Перейдём к методам преобразования и упорядочения массива.

#### map

Метод arr.map является одним из наиболее полезных и часто используемых.

Он вызывает функцию для каждого элемента массива и возвращает массив результатов выполнения этой функции.

Синтаксис:

*let result = arr.map(function(item, index, array) {*

*// возвращается новое значение вместо элемента*

*});*

Например, здесь мы преобразуем каждый элемент в его длину (Листинг 4.22):

**Листинг 4.22** Использование метода map

**let *lengths*** = [**"Bilbo"**, **"Gandalf"**, **"Nazgul"**].map(item => item.**length**);

*alert*(***lengths***); *// 5,7,6*

#### sort(fn)

Вызов arr.sort() сортирует массив на месте, меняя в нём порядок элементов.

Он возвращает отсортированный массив, но обычно возвращаемое значение игнорируется, так как изменяется сам arr (Листинг 4.23).

**Листинг 4.23** Использование метода sort

**let *arr*** = [ 1, 2, 15 ];

*// метод сортирует содержимое arr*

***arr***.sort();

*alert*( ***arr*** ); *// 1, 15, 2*

Не заметили ничего странного в этом примере?

Порядок стал 1, 15, 2. Это неправильно! Но почему?

**По умолчанию элементы сортируются как строки.**

Буквально, элементы преобразуются в строки при сравнении. Для строк применяется лексикографический порядок, и действительно выходит, что "2" > "15".

Чтобы использовать наш собственный порядок сортировки, нам нужно предоставить функцию в качестве аргумента arr.sort().

Функция должна для пары значений возвращать:

**function** *compare*(a, b) {

**if** (a > b) **return** 1; *// если первое значение больше второго*

**if** (a == b) **return** 0; *// если равны*

**if** (a < b) **return** -1; *// если первое значение меньше второго*

}

Например, для сортировки чисел код будет выглядеть так (Листинг 4.24).

**Листинг 4.24** Использование своей функции сортировки

**function** *compareNumeric*(a, b) {

**if** (a > b) **return** 1;

**if** (a == b) **return** 0;

**if** (a < b) **return** -1;

}

**let *arr*** = [ 1, 2, 15 ];

***arr***.sort(*compareNumeric*);

*alert*(***arr***); *// 1, 2, 15*

Теперь всё работает как надо.

Давайте возьмём паузу и подумаем, что же происходит. Упомянутый ранее массив arr может быть массивом чего угодно, верно? Он может содержать числа, строки, объекты или что-то ещё. У нас есть набор каких-то элементов. Чтобы отсортировать его, нам нужна функция, определяющая порядок, которая знает, как сравнивать его элементы. По умолчанию элементы сортируются как строки.

Метод arr.sort(fn) реализует общий алгоритм сортировки. Нам не нужно заботиться о том, как он работает внутри (в большинстве случаев это оптимизированная быстрая сортировка). Она проходится по массиву, сравнивает его элементы с помощью предоставленной функции и переупорядочивает их. Всё, что остаётся нам, это предоставить fn, которая делает это сравнение.

#### reverse

Метод arr.reverse меняет порядок элементов в arr на обратный.

Например (Листинг 4.25):

**Листинг 4.25** Использование функции reverse

**let *arr*** = [1, 2, 3, 4, 5];

***arr***.reverse();

*alert*( ***arr*** ); *// 5,4,3,2,1*

Он также возвращает массив arr с изменённым порядком элементов.

#### split и join

Ситуация из реальной жизни. Мы пишем приложение для обмена сообщениями, и посетитель вводит имена тех, кому его отправить, через запятую: Вася, Петя, Маша. Но нам-то гораздо удобнее работать с массивом имён, чем с одной строкой. Как его получить?

Метод str.split(delim) именно это и делает. Он разбивает строку на массив по заданному разделителю delim.

В примере ниже таким разделителем является строка из запятой и пробела (Листинг 4.26).

**Листинг 4.26** Использование функции split

**let *names*** = **'Вася, Петя, Маша'**;

**let *arr*** = ***names***.split(**', '**);

**for** (**let *name* of *arr***) {

*alert*( **`Сообщение получат:** ${***name***}**.`** ); *// Сообщение получат: Вася (и другие имена)*

}

У метода split есть необязательный второй числовой аргумент – ограничение на количество элементов в массиве. Если их больше, чем указано, то остаток массива будет отброшен.

Вызов arr.join(glue) делает в точности противоположное split. Он создаёт строку из элементов arr, вставляя glue между ними (Листинг 4.27).

**Листинг 4.27** Использование функции join

**let *arr*** = [**'Вася'**, **'Петя'**, **'Маша'**];

**let *str*** = ***arr***.join(**';'**); *// объединить массив в строку через ;*

*alert*( ***str*** ); *// Вася;Петя;Маша*

### Практическая работа №14

1. У нас есть массив строк arr. Нужно получить отсортированную копию, но оставить arr неизменённым.

Создайте функцию copySorted(arr), которая будет возвращать такую копию.

**let *arr*** = [**"HTML"**, **"JavaScript"**, **"CSS"**];

**let *sorted*** = copySorted(***arr***);

*alert*( ***sorted*** ); *// CSS, HTML, JavaScript*

*alert*( ***arr*** ); *// HTML, JavaScript, CSS (без изменений)*

1. Создайте функцию конструктор Calculator, которая создаёт «расширяемые» объекты калькулятора.

Задание состоит из двух частей.

1. Во-первых, реализуйте метод calculate(str), который принимает строку типа "1 + 2" в формате «ЧИСЛО оператор ЧИСЛО» (разделено пробелами) и возвращает результат. Метод должен понимать плюс + и минус -.

Пример использования:

**let *calc*** = **new** Calculator;

*alert*( ***calc***.calculate(**"3 + 7"**) ); *// 10*

1. Затем добавьте метод addMethod(name, func), который добавляет в калькулятор новые операции. Он принимает оператор name и функцию с двумя аргументами func(a,b), которая описывает его.

Например, давайте добавим умножение \*, деление / и возведение в степень \*\*:

**let *powerCalc*** = **new** Calculator;

***powerCalc***.addMethod(**"\*"**, (a, b) => a \* b);

***powerCalc***.addMethod(**"/"**, (a, b) => a / b);

***powerCalc***.addMethod(**"\*\*"**, (a, b) => a \*\* b);

**let *result*** = ***powerCalc***.calculate(**"2 \*\* 3"**);

*alert*( ***result*** ); *// 8*

Для этой задачи не нужны скобки или сложные выражения.

Числа и оператор разделены ровно одним пробелом.

Не лишним будет добавить обработку ошибок.

## Проверочная работа №2

Выполните предложенное ниже задание согласно своему варианту, составьте **отчет** с **блок-схемами** всех алгоритмов, **кодом** и **результатами работы**.

### Обобщенное задание

Выбрать предметную область для базы данных и предложить структуру объектов для описания отдельных записей базы данных. Выбранная структура должна иметь не менее пяти свойств двух и более типов. Для выбранной базы данных написать следующие функции:

1. Функцию формирования массива объектов, значения свойств которых вводятся с клавиатуры. При вводе объектов можно реализовать один из следующих механизмов:

* ввод заранее заданного количества объектов;
* ввод до появления объекта с заданным признаком;
* диалог с пользователем о необходимости продолжать ввод.

1. Функцию просмотра содержимого массива объектов.
2. Функцию дополнения уже существующего массива новыми объектами.
3. Функцию поиска и вывода на экран объекта (объектов) с заданным значением свойства.
4. Функцию упорядочения массива структур по заданному свойству.

### Варианты

1. «Человек»: фамилия; имя; отчество; пол; национальность; рост; вес; дата рождения (год, месяц число); номер телефона; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира).
2. «Школьник»: фамилия; имя; отчество; пол; национальность; рост; вес; дата рождения (год, месяц число); номер телефона; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); школа; класс.
3. «Студент»: фамилия; имя; отчество; пол; национальность; рост; вес; дата рождения (год, месяц число); номер телефона; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); ВУЗ; курс; группа; средний балл; специальность.
4. «Покупатель»: фамилия; имя; отчество; пол; национальность; рост; вес; дата рождения (год, месяц число); номер телефона; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); номер кредитной карточки; банковского счета.
5. «Пациент»: фамилия; имя; отчество; пол; национальность; рост; вес; дата рождения (год, месяц число); номер телефона; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); номер больницы; отделение; номер медицинской карты; диагноз; группа крови.
6. «Владелец автомобиля»: фамилия; имя; отчество; номер телефона; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира) марка автомобиля; номер автомобиля; номер техпаспорта.
7. «Военнослужащий»: фамилия; имя; отчество; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); национальность; дата рождения (год, месяц число); должность; звание.
8. «Рабочий»: фамилия; имя; отчество; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); национальность; дата рождения (год, месяц число); № цеха; табельный номер; образование; год поступления на работу.
9. «Владелец телефона»: фамилия; имя; отчество; домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); № телефона.
10. «Абитуриент»: фамилия; имя; отчество; пол; национальность; дата рождения (год, месяц число); домашний адрес (почтовый индекс, страна, область, район, город, улица, дом, квартира); оценки по экзаменам; проходной балл.
11. «Государство»: название страны; столица; государственный язык; население; площадь территории; денежная единица; государственный строй; глава государства.
12. «Автомобиль»: марка; цвет; серийный номер; регистрационный номер; год выпуска; год техосмотра; цена.
13. «Рейс»: марка автомобиля; номер автомобиля; пункт назначения; грузоподъемность (в тоннах); стоимость единицы груза; общая стоимость груза.
14. «Книга»: название; автор (фамилия; имя); год выхода; издательство; себестоимость; цена; прибыль.
15. «Здание»: адрес; тип здания; количество этажей; количество квартир; срок эксплуатации; срок до капитального ремонта (25 лет - срок эксплуатации).

### Критерии оценивания

3: Изучен теоретический материал 3-4 главы. Выполнены задания (возможно реализация не всех функций). Составлен отчет о работе;

4: Изучен теоретический материал 3-4 главы. Выполнены задания (возможно с незначительными погрешностями). Составлен отчет о работе;

5: Изучен теоретический материал 3-4 главы. Выполнены задания. Составлен отчет о работе. Присутствует единый стиль оформления кода, комментарии.

# 5. Документ и объекты страницы

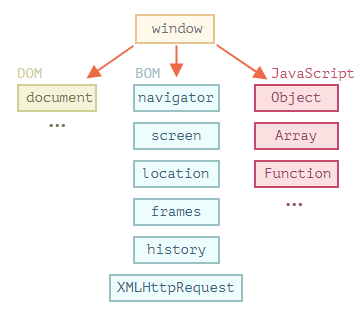
## Браузерное окружение

Язык JavaScript изначально был создан для веб-браузеров. Но с тех пор он значительно эволюционировал и превратился в кроссплатформенный язык программирования для решения широкого круга задач.

Сегодня JavaScript может использоваться в браузере, на веб-сервере или в какой-то другой среде, даже в кофеварке. Каждая среда предоставляет свой функционал, который спецификация JavaScript называет *окружением*.

Окружение предоставляет свои объекты и дополнительные функции, в дополнение базовым языковым. Браузеры, например, дают средства для управления веб-страницами. Node.js делает доступными какие-то серверные возможности и так далее.

На Рисунке 5.1 в общих чертах показано, что доступно для JavaScript в браузерном окружении.



**Рисунок 5.1.** Браузерное окружение

Как мы видим, имеется корневой объект window, который выступает в 2 ролях:

1. Во-первых, это глобальный объект для JavaScript-кода.
2. Во-вторых, он также представляет собой окно браузера и располагает методами для управления им.

Например, здесь мы используем window как глобальный объект:

**function** *sayHi*() {

*alert*(**"Hello"**);

}

*// глобальные функции доступны как методы глобального объекта:*

***window***.*sayHi*();

А здесь мы используем window как объект окна браузера, чтобы узнать его высоту:

*alert*(***window***.**innerHeight**); *// внутренняя высота окна браузера*

## DOM (Document Object Model)

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

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

Например (Листинг 5.1):

**Листинг 5.1** Использование document

*// заменим цвет фона на красный,*

***document***.**body**.**style**.**background** = **"red"**;

*// а через секунду вернём как было*

*setTimeout*(() => ***document***.**body**.**style**.**background** = **""**, 1000);

Мы использовали в примере только document.body.style, но на самом деле возможности по управлению страницей намного шире. Различные свойства и методы описаны в спецификации: **DOM Living Standard на https://dom.spec.whatwg.org**

## BOM (Browser Object Model)

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

Например:

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

Вот как мы можем использовать объект location (Листинг 5.2):

**Листинг 5.2** Использование location

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

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

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

}

## DOM-дерево

Основой HTML-документа являются теги.

В соответствии с объектной моделью документа («Document Object Model», коротко DOM), каждый HTML-тег является объектом. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом.

Все эти объекты доступны при помощи JavaScript, мы можем использовать их для изменения страницы.

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

### Пример DOM

Начнём с такого, простого, документа:

<!DOCTYPE **HTML**>

<**html**>

<**head**>

<**title**>О лосях</**title**>

</**head**>

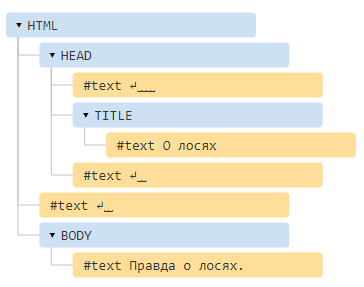
<**body**>

Правда о лосях.

</**body**>

</**html**>

DOM – это представление HTML-документа в виде дерева тегов (Рисунок 5.2).



**Рисунок 5.2.** DOM представление HTML-документа

На рисунке выше узлы-элементы можно кликать, и их дети будут скрываться и раскрываться.

Каждый узел этого дерева – это объект.

Теги являются *узлами-элементами* (или просто элементами). Они образуют структуру дерева: <html> – это корневой узел, <head> и <body> его дочерние узлы и т.д.

Текст внутри элементов образует *текстовые узлы*, обозначенные как #text. Текстовый узел содержит в себе только строку текста. У него не может быть потомков, т.е. он находится всегда на самом нижнем уровне.

Например, в теге <title> есть текстовый узел "О лосях".

Обратите внимание на специальные символы в текстовых узлах:

* перевод строки: ↵ (в JavaScript он обозначается как \n)
* пробел: ␣

Пробелы и переводы строки – это полноправные символы, как буквы и цифры. Они образуют текстовые узлы и становятся частью дерева DOM. Так, в примере выше в теге <head> есть несколько пробелов перед <title>, которые образуют текстовый узел #text (он содержит в себе только перенос строки и несколько пробелов).

### Другие типы узлов

Есть и некоторые другие типы узлов, кроме элементов и текстовых узлов.

Например, узел-комментарий:

<!DOCTYPE **HTML**>

<**html**>

<**body**>

Правда о лосях.

<**ol**>

<**li**>Лось -- животное хитрое</**li**>

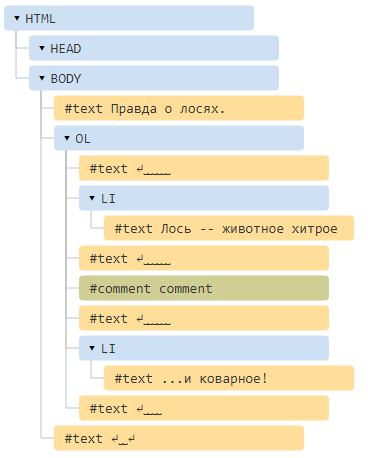
*<!-- комментарий -->*

<**li**>...и коварное!</**li**>

</**ol**>

</**body**>

</**html**>



**Рисунок 5.3.** DOM представление комментария

Здесь мы видим узел нового типа – комментарий, обозначенный как #comment, между двумя текстовыми узлами.

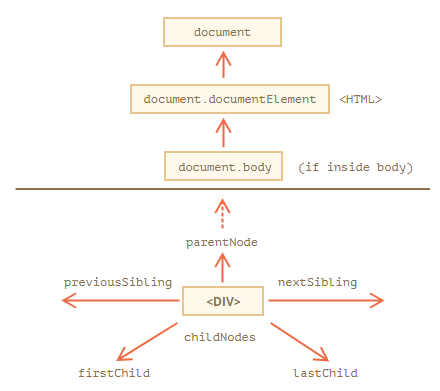
**Все, что есть в HTML, даже комментарии, является частью DOM.**

## Навигация по DOM-элементам

DOM позволяет нам делать что угодно с элементами и их содержимым, но для начала нужно получить соответствующий DOM-объект.

Все операции с DOM начинаются с объекта document. Это главная «точка входа» в DOM. Из него мы можем получить доступ к любому узлу.

Так выглядят основные ссылки, по которым можно переходить между узлами DOM (Рисунок 5.4).



**Рисунок 5.4.** Перемещение по DOM

### Сверху: documentElement и body

Самые верхние элементы дерева доступны как свойства объекта document:

**<html> = document.documentElement**

Самый верхний узел документа: document.documentElement. В DOM он соответствует тегу <html>.

**<body> = document.body**

Другой часто используемый DOM-узел – узел тега <body>: document.body.

**<head> = document.head**

Тег <head> доступен как document.head.

### Дети: childNodes, firstChild, lastChild

Здесь и далее мы будем использовать два принципиально разных термина:

* **Дочерние узлы (или дети)** – элементы, которые являются непосредственными детьми узла. Другими словами, элементы, которые лежат непосредственно внутри данного. Например, <head> и <body> являются детьми элемента <html>.
* **Потомки** – все элементы, которые лежат внутри данного, включая детей, их детей и т.д.

**Коллекция childNodes содержит список всех детей, включая текстовые узлы.**

Пример ниже последовательно выведет детей document.body (Листинг 5.3):

**Листинг 5.3** Коллекция childNodes

<**html**>

<**body**>

<**div**>Начало</**div**>

<**ul**>

<**li**>Информация</**li**>

</**ul**>

<**div**>Конец</**div**>

<**script**>

**for** (**let *i*** = 0; ***i*** < ***document***.**body**.**childNodes**.**length**; ***i***++) {

*alert*( ***document***.**body**.**childNodes**[***i***] ); *// Text, DIV, Text, UL, ..., SCRIPT*

}

</**script**>

...какой-то HTML-код...

</**body**>

</**html**>

Обратим внимание на маленькую деталь. Если запустить пример выше, то последним будет выведен элемент <script>. На самом деле, в документе есть ещё «какой-то HTML-код», но на момент выполнения скрипта браузер ещё до него не дошёл, поэтому скрипт не видит его.

**Свойства firstChild и lastChild обеспечивают быстрый доступ к первому и последнему дочернему элементу.**

Для проверки наличия дочерних узлов существует также специальная функция elem.hasChildNodes().

### DOM-коллекции

Как мы уже видели, childNodes похож на массив. На самом деле это не массив, а коллекция – особый перебираемый объект-псевдомассив.

И есть два важных следствия из этого:

* Для перебора коллекции мы можем использовать for..of;
* Методы массивов не будут работать, потому что коллекция – это не массив.

### Соседи и родитель

*Соседи* – это узлы, у которых один и тот же родитель.

Например, здесь <head> и <body> соседи:

<**html**>

<**head**>...</**head**><**body**>...</**body**>

</**html**>

* говорят, что <body> – «следующий» или «правый» сосед <head>
* также можно сказать, что <head> «предыдущий» или «левый» сосед <body>.

Следующий узел того же родителя (следующий сосед) – в свойстве nextSibling, а предыдущий – в previousSibling.

Родитель доступен через parentNode (Листинг 5.4).

**Листинг 5.4** Соседские элементы

*// родителем <body> является <html>*

*alert*( ***document***.**body**.**parentNode** === ***document***.**documentElement** ); *// выведет true*

*// после <head> идёт <body>*

*alert*( ***document***.**head**.**nextSibling** ); *// HTMLBodyElement*

*// перед <body> находится <head>*

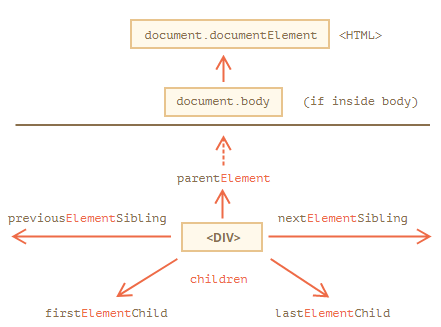
*alert*( ***document***.**body**.**previousSibling** ); *// HTMLHeadElement*

### Навигация только по элементам

Навигационные свойства, описанные выше, относятся ко всем узлам в документе. В частности, в childNodes находятся и текстовые узлы и узлы-элементы и узлы-комментарии, если они есть.

Но для большинства задач текстовые узлы и узлы-комментарии нам не нужны. Мы хотим манипулировать узлами-элементами, которые представляют собой теги и формируют структуру страницы.

Поэтому давайте рассмотрим дополнительный набор ссылок, которые учитывают только узлы-элементы (Рисунок 5.5):



**Рисунок 5.4.** Перемещение по узлам-элементам

Эти ссылки похожи на те, что раньше, только в ряде мест стоит слово Element:

* children – коллекция детей, которые являются элементами.
* firstElementChild, lastElementChild – первый и последний дочерний элемент.
* previousElementSibling, nextElementSibling – соседи-элементы.
* parentElement – родитель-элемент.

### Практическая работа №15

1. Для страницы:

<**html**>

<**body**>

<**div**>Пользователи:</**div**>

<**ul**>

<**li**>Джон</**li**>

<**li**>Пит</**li**>

</**ul**>

</**body**>

</**html**>

* Напишите код, который получит элемент <div>
* Напишите код, который получит <ul>
* Напишите код, который получит второй <li> (с именем Пит)

## Поиск: getElement\*, querySelector\*

Свойства навигации по DOM хороши, когда элементы расположены рядом. А что, если нет? Как получить произвольный элемент страницы?

Для этого в DOM есть дополнительные методы поиска.

### document.getElementById или просто id

Если у элемента есть атрибут id, то мы можем получить его вызовом document.getElementbyId(id), где бы он ни находился (Листинги 5.5, 5.6).

**Листинг 5.5** Использование getElementbyId

<**div id="elem"**>

<**div id="elem-content"**>Element</**div**>

</**div**>

<**script**>

*// получить элемент*

**let *elem*** = ***document***.getElementById(**'elem'**);

*// сделать его фон красным*

***elem***.**style**.**background** = **'red'**;

</**script**>

**Листинг 5.6** Использование глобальной переменной

<**div id="elem"**>

<**div id="elem-content"**>Элемент</**div**>

</**div**>

<**script**>

*// elem - ссылка на элемент с id="elem"*

elem.**style**.**background** = **'red'**;

*// внутри id="elem-content" есть дефис, так что такой id не может служить именем переменной*

*// ...но мы можем обратиться к нему через квадратные скобки: window['elem-content']*

</**script**>

**Пожалуйста, не используйте такие глобальные переменные для доступа к элементам**

**Это поведение соответствует стандарту, но поддерживается в основном для совместимости, как осколок далекого прошлого.**

### querySelectorAll

Самый универсальный метод поиска – это elem.querySelectorAll(css), он возвращает все элементы внутри elem, удовлетворяющие данному CSS-селектору.

Следующий запрос получает все элементы <li>, которые являются последними потомками в <ul> (Листинг 5.7).

**Листинг 5.7** Использование querySelectorAll

<**ul**>

<**li**>Этот</**li**>

<**li**>тест</**li**>

</**ul**>

<**ul**>

<**li**>полностью</**li**>

<**li**>пройден</**li**>

</**ul**>

<**script**>

**let *elements*** = ***document***.querySelectorAll(**'ul > li:last-child'**);

**for** (**let *elem* of *elements***) {

*alert*(***elem***.**innerHTML**); *// "тест", "пройден"*

}

</**script**>

Этот метод действительно мощный, потому что можно использовать любой CSS-селектор.

### querySelector

Метод elem.querySelector(css) возвращает первый элемент, соответствующий данному CSS-селектору.

Иначе говоря, результат такой же, как при вызове elem.querySelectorAll(css)[0], но он сначала найдёт все элементы, а потом возьмёт первый, в то время как elem.querySelector найдёт только первый и остановится. Это быстрее, кроме того, его короче писать.

### getElementsBy\*

Существуют также другие методы поиска элементов по тегу, классу и так далее.

На данный момент, они скорее исторические, так как querySelector более чем эффективен.

Здесь мы рассмотрим их для полноты картины, также вы можете встретить их в старом коде.

* elem.getElementsByTagName(tag) ищет элементы с данным тегом и возвращает их коллекцию. Передав "\*" вместо тега, можно получить всех потомков.
* elem.getElementsByClassName(className) возвращает элементы, которые имеют данный CSS-класс.
* document.getElementsByName(name) возвращает элементы с заданным атрибутом name. Очень редко используется.

## Свойства узлов: тип, тег и содержимое

Теперь давайте более внимательно взглянем на DOM-узлы.

В этой главе мы подробнее разберём, что они собой представляют и изучим их основные свойства.

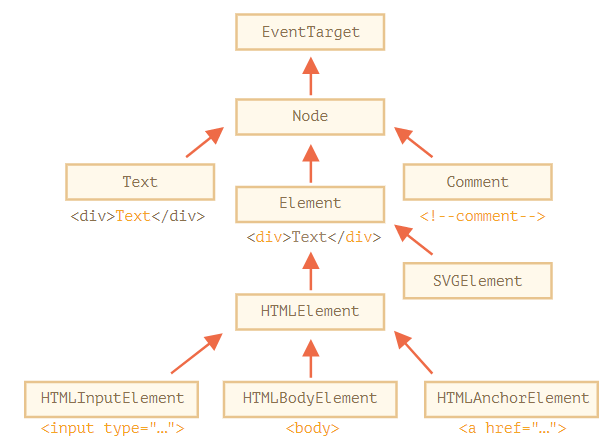
### Классы DOM-узлов

У разных DOM-узлов могут быть разные свойства. Например, у узла, соответствующего тегу <a>, есть свойства, связанные со ссылками, а у соответствующего тегу <input> – свойства, связанные с полем ввода и т.д. Текстовые узлы отличаются от узлов-элементов. Но у них есть общие свойства и методы, потому что все классы DOM-узлов образуют единую иерархию.

Каждый DOM-узел принадлежит соответствующему встроенному классу.

Корнем иерархии является EventTarget, от него наследует Node и остальные DOM-узлы.

На рисунке ниже изображены основные классы (Рисунок 5.5):



**Рисунок 5.5.** Иерархия DOM

Существуют следующие классы:

* *EventTarget* – это корневой «абстрактный» класс. Объекты этого класса никогда не создаются. Он служит основой, благодаря которой все DOM-узлы поддерживают так называемые «события», о которых мы поговорим позже.
* *Node* – также является «абстрактным» классом, и служит основой для DOM-узлов. Он обеспечивает базовую функциональность: parentNode, nextSibling, childNodes и т.д. (это геттеры). Объекты класса Node никогда не создаются. Но есть определённые классы узлов, которые наследуют от него: Text – для текстовых узлов, Element – для узлов-элементов и более экзотический Comment – для узлов-комментариев.
* *Element* – это базовый класс для DOM-элементов. Он обеспечивает навигацию на уровне элементов: nextElementSibling, children и методы поиска: getElementsByTagName, querySelector. Браузер поддерживает не только HTML, но также XML и SVG. Класс Element служит базой для следующих классов: SVGElement, XMLElement и HTMLElement.
* *HTMLElement* – является базовым классом для всех остальных HTML-элементов. От него наследуют конкретные элементы:
  + HTMLInputElement – класс для тега <input>,
  + HTMLBodyElement – класс для тега <body>,
  + HTMLAnchorElement – класс для тега <a>,
  + …и т.д, каждому тегу соответствует свой класс, который предоставляет определённые свойства и методы.

### Тег: nodeName и tagName

Получив DOM-узел, мы можем узнать имя его тега из свойств nodeName и tagName:

Например:

*alert*( ***document***.**body**.**nodeName** ); *// BODY*

*alert*( ***document***.**body**.**tagName** ); *// BODY*

Есть ли какая-то разница между tagName и nodeName?

Да, она отражена в названиях свойств, но не очевидна.

* Свойство tagName есть только у элементов Element.
* Свойство nodeName определено для любых узлов Node:
  + для элементов оно равно tagName.
  + для остальных типов узлов (текст, комментарий и т.д.) оно содержит строку с типом узла.

### innerHTML: содержимое элемента

Свойство innerHTML позволяет получить HTML-содержимое элемента в виде строки.

Мы также можем изменять его. Это один из самых мощный способов менять содержимое на странице.

Пример ниже показывает содержимое document.body, а затем полностью заменяет его (Листинг 5.8):

**Листинг 5.8** Использование innerHTML

<**body**>

<**p**>Параграф</**p**>

<**div**>DIV</**div**>

<**script**>

*alert*( ***document***.**body**.**innerHTML** ); *// читаем текущее содержимое*

***document***.**body**.**innerHTML** = **'Новый BODY!'**; *// заменяем содержимое*

</**script**>

</**body**>

**Будьте внимательны: «innerHTML+=» осуществляет перезапись. Так как содержимое «обнуляется» и переписывается заново, все изображения и другие ресурсы будут перезагружены.**

### outerHTML: HTML элемента целиком

Свойство outerHTML содержит HTML элемента целиком. Это как innerHTML плюс сам элемент.

Посмотрим на пример (Листинг 5.9):

**Листинг 5.9** Использование outerHTML

<**div id="elem"**>Привет <**b**>Мир</**b**></**div**>

<**script**>

*alert*(elem.**outerHTML**); *// <div id="elem">Привет <b>Мир</b></div>*

</**script**>

**Будьте осторожны: в отличие от innerHTML, запись в outerHTML не изменяет элемент. Вместо этого элемент заменяется целиком во внешнем контексте.**

Мы можем писать в elem.outerHTML, но надо иметь в виду, что это не меняет элемент, в который мы пишем. Вместо этого создаётся новый HTML на его месте. Мы можем получить ссылки на новые элементы, обратившись к DOM.

### nodeValue/data: содержимое текстового узла

Свойство innerHTML есть только у узлов-элементов.

У других типов узлов, в частности, у текстовых, есть свои аналоги: свойства nodeValue и data. Эти свойства очень похожи при использовании, есть лишь небольшие различия в спецификации. Мы будем использовать data, потому что оно короче.

Прочитаем содержимое текстового узла и комментария (Листинг 5.11):

**Листинг 5.11** Использование data

<**body**>

Привет

*<!-- Комментарий -->*

<**script**>

**let *text*** = ***document***.**body**.**firstChild**;

*alert*(***text***.**data**); *// Привет*

**let *comment*** = ***text***.**nextSibling**;

*alert*(***comment***.**data**); *// Комментарий*

</**script**>

</**body**>

Мы можем представить, для чего нам может понадобиться читать или изменять текстовый узел, но комментарии?

Иногда их используют для вставки информации и инструкций шаблонизатора в HTML.

### textContent: просто текст

Свойство textContent предоставляет доступ к тексту внутри элемента за вычетом всех <тегов>.

Например (Листинг 5.12):

**Листинг 5.12** Использование data

<**div id="news"**>

<**h1**>Срочно в номер!</**h1**>

<**p**>Марсиане атаковали человечество!</**p**>

</**div**>

<**script**>

*// Срочно в номер! Марсиане атаковали человечество!*

*alert*(news.**textContent**);

</**script**>

Как мы видим, возвращается только текст, как если бы все <теги> были вырезаны, но текст в них остался.

На практике редко появляется необходимость читать текст таким образом.

**Намного полезнее возможность записывать текст в textContent, т.к. позволяет писать текст «безопасным способом».**

### Свойство «hidden»

Атрибут и DOM-свойство «hidden» указывает на то, видим ли мы элемент или нет.

Мы можем использовать его в HTML или назначать при помощи JavaScript, как в примере ниже (Листинг 5.13):

**Листинг 5.13** Использование hidden

<**div**>Оба тега DIV внизу невидимы</**div**>

<**div hidden**>С атрибутом "hidden"</**div**>

<**div id="elem"**>С назначенным JavaScript свойством "hidden"</**div**>

<**script**>

elem.**hidden** = **true**;

</**script**>

Технически, hidden работает так же, как style="display:none". Но его применение проще.

Мигающий элемент (Листинг 5.14):

**Листинг 5.14** Мигающий элемент

<**div id="elem"**>Мигающий элемент</**div**>

<**script**>

*setInterval*(() => elem.**hidden** = !elem.**hidden**, 1000);

</**script**>

### Другие свойства

У DOM-элементов есть дополнительные свойства, в частности, зависящие от класса:

* value – значение для <input>, <select> и <textarea> (HTMLInputElement, HTMLSelectElement…).
* href – адрес ссылки «href» для <a href="..."> (HTMLAnchorElement).
* id – значение атрибута «id» для всех элементов (HTMLElement).
* …и многие другие…

Большинство стандартных HTML-атрибутов имеют соответствующее DOM-свойство, и мы можем получить к нему доступ.

Если мы хотим узнать полный список поддерживаемых свойств для данного класса, можно найти их в спецификации. Например, класс HTMLInputElement описывается здесь: https://html.spec.whatwg.org/#htmlinputelement.

Если же нам нужно быстро что-либо узнать или нас интересует специфика определённого браузера – мы всегда можем вывести элемент в консоль, используя console.dir(elem), и прочитать все свойства. Или исследовать «свойства DOM» во вкладке Elements браузерных инструментов разработчика.

### Практическая работа №16

1. У нас есть дерево, структурированное как вложенные списки ul/li.

Напишите код, который выведет каждый элемент списка <li>:

* Какой в нём текст (без поддерева) ?
* Какое число потомков – всех вложенных <li> (включая глубоко вложенные) ?

## Атрибуты и свойства

Когда браузер загружает страницу, он «читает» (также говорят: «парсит») HTML и генерирует из него DOM-объекты. Для узлов-элементов большинство стандартных HTML-атрибутов автоматически становятся свойствами DOM-объектов.

Например, для такого тега <body id="page"> у DOM-объекта будет такое свойство body.id="page".

Но преобразование атрибута в свойство происходит не один-в-один! В этой главе мы уделим внимание различию этих двух понятий, чтобы посмотреть, как работать с ними, когда они одинаковые и когда разные.

### DOM-свойства

Ранее мы уже видели встроенные DOM-свойства. Их много. Но технически нас никто не ограничивает, и если этого мало – мы можем добавить своё собственное свойство.

DOM-узлы – это обычные объекты JavaScript. Мы можем их изменять.

Например, создадим новое свойство для document.body (Листинг 5.15):

**Листинг 5.15** Добавление собственных свойств

***document***.**body**.**myData** = {

**name**: **'Caesar'**,

**title**: **'Imperator'**

};

*alert*(***document***.**body**.myData.**title**); *// Imperator*

Итак, DOM-свойства и методы ведут себя так же, как и обычные объекты JavaScript:

* Им можно присвоить любое значение.
* Они регистрозависимы (нужно писать elem.nodeType, не elem.NoDeTyPe).

### HTML-атрибуты

В HTML у тегов могут быть атрибуты. Когда браузер парсит HTML, чтобы создать DOM-объекты для тегов, он распознает *стандартные* атрибуты и создает DOM-свойства для них.

Таким образом, когда у элемента есть id или другой стандартный атрибут, создается соответствующее свойство. Но этого не происходит, если атрибут нестандартный.

Таким образом, для нестандартных атрибутов не будет соответствующих DOM-свойств. Есть ли способ получить такие атрибуты?

Конечно. Все атрибуты доступны с помощью следующих методов:

* elem.hasAttribute(name) – проверяет наличие атрибута.
* elem.getAttribute(name) – получает значение атрибута.
* elem.setAttribute(name, value) – устанавливает значение атрибута.
* elem.removeAttribute(name) – удаляет атрибут.

У HTML-атрибутов есть следующие особенности:

* Их имена регистронезависимы (id то же самое, что и ID).
* Их значения всегда являются строками.

### Синхронизация между атрибутами и свойствами

Когда стандартный атрибут изменяется, соответствующее свойство автоматически обновляется. Это работает и в обратную сторону (за некоторыми исключениями).

### DOM-свойства типизированы

DOM-свойства не всегда являются строками. Например, свойство input.checked (для чекбоксов) имеет логический тип (Листинг 5.16):

**Листинг 5.16** Типизация DOM-свойств

<**input id="input" type="checkbox" checked**> checkbox

<**script**>

*alert*(input.getAttribute(**'checked'**)); *// значение атрибута: пустая строка*

*alert*(input.**checked**); *// значение свойства: true*

</**script**>

Хотя большинство свойств, всё же, строки.

При этом некоторые из них, хоть и строки, могут отличаться от атрибутов. Например, DOM-свойство href всегда содержит полный URL, даже если атрибут содержит относительный URL или просто #hash.

### Практическая работа №17

1. Сделайте все внешние ссылки оранжевыми, изменяя их свойство style.

Ссылка является внешней, если:

* Её href содержит ://
* Но не начинается с http://internal.com.

Пример:

<**a name="list"**>the list</**a**>

<**ul**>

<**li**><**a href="http://google.com"**>http://google.com</**a**></**li**>

<**li**><**a href="/tutorial"**>/tutorial.html</**a**></**li**>

<**li**><**a href="local/path"**>local/path</**a**></**li**>

<**li**><**a href="ftp://ftp.com/my.zip"**>ftp://ftp.com/my.zip</**a**></**li**>

<**li**><**a href="http://nodejs.org"**>http://nodejs.org</**a**></**li**>

<**li**><**a href="http://internal.com/test"**>http://internal.com/test</**a**></**li**>

</**ul**>

<**script**>

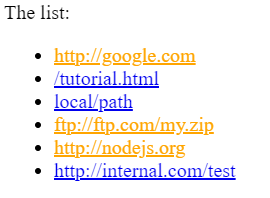
*// добавление стиля для одной ссылки*

**let *link*** = ***document***.querySelector(**'a'**);

***link***.**style**.**color** = **'orange'**;

</**script**>

Результат должен быть таким):



## Изменение документа

Модификации DOM – это ключ к созданию «живых» страниц.

Здесь мы увидим, как создавать новые элементы «на лету» и изменять уже существующие.

### Пример: показать сообщение

Рассмотрим методы на примере – а именно, добавим на страницу сообщение, которое будет выглядеть получше, чем alert.

Вот такое (Листинг 5.17, Рисунок 5.):

**Листинг 5.17** Верстка для всплывающего сообщения

<**style**>

.**alert** {

**padding**: 15**px**;

**border**: 1**px solid #d6e9c6**;

**border-radius**: 4**px**;

**color**: **#3c763d**;

**background-color**: **#dff0d8**;

}

</**style**>

<**div class="alert"**>

<**strong**>Всем привет!</**strong**> Вы прочитали важное сообщение.

</**div**>



**Рисунок 5.6.** Информационное сообщение

Это был пример HTML. Теперь давайте создадим такой же div, используя JavaScript (предполагаем, что стили в HTML или во внешнем CSS-файле).

### Создание элемента

DOM-узел можно создать двумя методами:

**document.createElement(tag)**

Создаёт новый элемент с заданным тегом:

**let *div*** = ***document***.createElement(**'div'**);

**document.createTextNode(text)**

Создаёт новый текстовый узел с заданным текстом:

**let *textNode*** = ***document***.createTextNode(**'А вот и я'**);

В нашем случае сообщение – это div с классом alert и HTML в нём (Листинг 5.18).

**Листинг 5.18** Создание сообщения

**let *div*** = ***document***.createElement(**'div'**);

***div***.**className** = **"alert"**;

***div***.**innerHTML** = **"<strong>Всем привет!</strong> Вы прочитали важное сообщение."**;

Мы создали элемент, но пока он только в переменной. Мы не можем видеть его на странице, поскольку он не является частью документа.

### Методы вставки

Чтобы наш div появился, нам нужно вставить его где-нибудь в document. Например, в document.body.

Для этого есть метод append, в нашем случае: document.body.append(div).

Вот полный пример (Листинг 5.19).

**Листинг 5.19** Создание сообщения. Полный вариант.

<**style**>

.**alert** {

**padding**: 15**px**;

**border**: 1**px solid #d6e9c6**;

**border-radius**: 4**px**;

**color**: **#3c763d**;

**background-color**: **#dff0d8**;

}

</**style**>

<**script**>

**let *div*** = ***document***.createElement(**'div'**);

***div***.**className** = **"alert"**;

***div***.**innerHTML** = **"<strong>Всем привет!</strong> Вы прочитали важное сообщение."**;

***document***.**body**.append(***div***);

</**script**>

Вот методы для различных вариантов вставки:

* node.append(...nodes or strings) – добавляет узлы или строки в конец node,
* node.prepend(...nodes or strings) – вставляет узлы или строки в начало node,
* node.before(...nodes or strings) –- вставляет узлы или строки до node,
* node.after(...nodes or strings) –- вставляет узлы или строки после node,
* node.replaceWith(...nodes or strings) –- заменяет node заданными узлами или строками.

### Удаление узлов

Для удаления узла есть методы node.remove().

Например, сделаем так, чтобы наше сообщение удалялось через секунду (Листинг 5.20)

**Листинг 5.20** Удаление сообщения

<**style**>

.**alert** {

**padding**: 15**px**;

**border**: 1**px solid #d6e9c6**;

**border-radius**: 4**px**;

**color**: **#3c763d**;

**background-color**: **#dff0d8**;

}

</**style**>

<**script**>

**let *div*** = ***document***.createElement(**'div'**);

***div***.**className** = **"alert"**;

***div***.**innerHTML** = **"<strong>Всем привет!</strong> Вы прочитали важное сообщение."**;

***document***.**body**.append(***div***);

*setTimeout*(() => ***div***.remove(), 1000);

</**script**>

Если нам нужно переместить элемент в другое место – нет необходимости удалять его со старого.

**Все методы вставки автоматически удаляют узлы со старых мест.**

Например, давайте поменяем местами элементы (Листинг 5.21).

**Листинг 5.21** Перемена мест двух элементов

<**div id="first"**>Первый</**div**>

<**div id="second"**>Второй</**div**>

<**script**>

*// нет необходимости вызывать метод remove*

second.after(first); *// берёт #second и после него вставляет #first*

</**script**>

### Клонирование узлов: cloneNode

Как вставить ещё одно подобное сообщение?

Мы могли бы создать функцию и поместить код туда. Альтернатива – клонировать существующий div и изменить текст внутри него (при необходимости).

Иногда, когда у нас есть большой элемент, это может быть быстрее и проще.

* Вызов elem.cloneNode(true) создаёт «глубокий» клон элемента – со всеми атрибутами и дочерними элементами. Если мы вызовем elem.cloneNode(false), тогда клон будет без дочерних элементов.

### Практическая работа №18

1. Создайте функцию clear(elem), которая удаляет всё содержимое из elem.

<**ol id="elem"**>

<**li**>Привет</**li**>

<**li**>Мир</**li**>

</**ol**>

<**script**>

**function** *clear*(elem) { */\* ваш код \*/* }

*clear*(elem); *// очищает список*

</**script**>

1. Напишите интерфейс для создания списка.

Для каждого пункта:

* Запрашивайте содержимое пункта у пользователя с помощью prompt.
* Создавайте элемент <li> и добавляйте его к <ul>.
* Процесс прерывается, когда пользователь нажимает Esc или вводит пустую строку.

Все элементы должны создаваться динамически.

Если пользователь вводит HTML-теги -– пусть в списке они показываются как обычный текст.

1. Создайте цветные часы как в примере ниже:



Для стилизации используйте HTML/CSS, JavaScript должен только обновлять время в элементах.

P.S: Используйте функцию setInterval

## Стили и классы

До того, как начнёте изучать способы работы со стилями и классами в JavaScript, есть одно важное правило. Надеемся, это достаточно очевидно, но мы все равно должны об этом упомянуть.

Как правило, существует два способа задания стилей для элемента:

1. Создать класс в CSS и использовать его: <div class="...">
2. Писать стили непосредственно в атрибуте style: <div style="...">.

JavaScript может менять и классы, и свойство style.

Классы – всегда предпочтительный вариант по сравнению со style. Мы должны манипулировать свойством style только в том случае, если классы «не могут справиться».

Например, использование style является приемлемым, если мы вычисляем координаты элемента динамически и хотим установить их из JavaScript.

В других случаях, например, чтобы сделать текст красным, добавить значок фона – описываем это в CSS и добавляем класс (JavaScript может это сделать). Это более гибкое и лёгкое в поддержке решение.

### className и classList

Изменение класса является одним из наиболее часто используемых действий в скриптах.

Когда-то давно в JavaScript существовало ограничение: зарезервированное слово типа "class" не могло быть свойством объекта. Это ограничение сейчас отсутствует, но в то время было невозможно иметь свойство elem.class.

Поэтому для классов было введено схожее свойство "className": elem.className соответствует атрибуту "class" (Листинг 5.22).

**Листинг 5.22** Использование className

<**body class="main page"**>

<**script**>

*alert*(***document***.**body**.**className**); *// main page*

</**script**>

</**body**>

Если мы присваиваем что-то elem.className, то это заменяет всю строку с классами. Иногда это то, что нам нужно, но часто мы хотим добавить/удалить один класс.

Для этого есть другое свойство: elem.classList.

elem.classList – это специальный объект с методами для добавления/удаления одного класса.

Например (Листинг 5.23):

**Листинг 5.23** Использование classList

<**body class="main page"**>

<**script**>

*// добавление класса*

***document***.**body**.**classList**.add(**'article'**);

*alert*(***document***.**body**.**className**); *// main page article*

</**script**>

</**body**>

Так что мы можем работать как со строкой полного класса, используя className, так и с отдельными классами, используя classList. Выбираем тот вариант, который нам удобнее.

Методы classList:

* elem.classList.add/remove("class") – добавить/удалить класс.
* elem.classList.toggle("class") – добавить класс, если его нет, иначе удалить.
* elem.classList.contains("class") – проверка наличия класса, возвращает true/false.

Кроме того, classList является перебираемым, поэтому можно перечислить все классы при помощи for..of.

### Element style

Свойство elem.style – это объект, который соответствует тому, что написано в атрибуте "style". Установка стиля elem.style.width="100px" работает так же, как наличие в атрибуте style строки width:100px.

Для свойства из нескольких слов используется camelCase:

background-color => elem.style.backgroundColor

z-index => elem.style.zIndex

border-left-width => elem.style.borderLeftWidth

Например:

***document***.**body**.**style**.**backgroundColor** = *prompt*(**'background color?'**, **'green'**);

### Практическая работа №19

1. Напишите функцию showNotification(options), которая создаёт уведомление: <div class="notification"> с заданным содержимым. Уведомление должно автоматически исчезнуть через 1,5 секунды.

Пример объекта options:

*// показывает элемент с текстом "Hello" рядом с правой верхней частью окна.*

showNotification({

**top**: 10, *// 10px от верхней границы окна (по умолчанию 0px)*

**right**: 10, *// 10px от правого края окна (по умолчанию 0px)*

**html**: **"Hello!"**, *// HTML-уведомление*

**className**: **"welcome"** *// дополнительный класс для div (необязательно)*

});

Используйте CSS-позиционирование для отображения элемента в заданных координатах. Исходный документ имеет необходимые стили.

## Размеры и прокрутка элементов

Существует множество JavaScript-свойств, которые позволяют считывать информацию об элементе: ширину, высоту и другие геометрические характеристики. В этой главе мы будем называть их «метрики».

Они часто требуются, когда нам нужно передвигать или позиционировать элементы с помощью JavaScript.

В качестве простого примера демонстрации свойств мы будем использовать следующий элемент (Листинг 5.24).

**Листинг 5.24** Пример элемента

<**div id="example"**>

...Текст...

</**div**>

<**style**>

**#example** {

**width**: 300**px**;

**height**: 200**px**;

**border**: 25**px solid #E8C48F**;

**padding**: 20**px**;

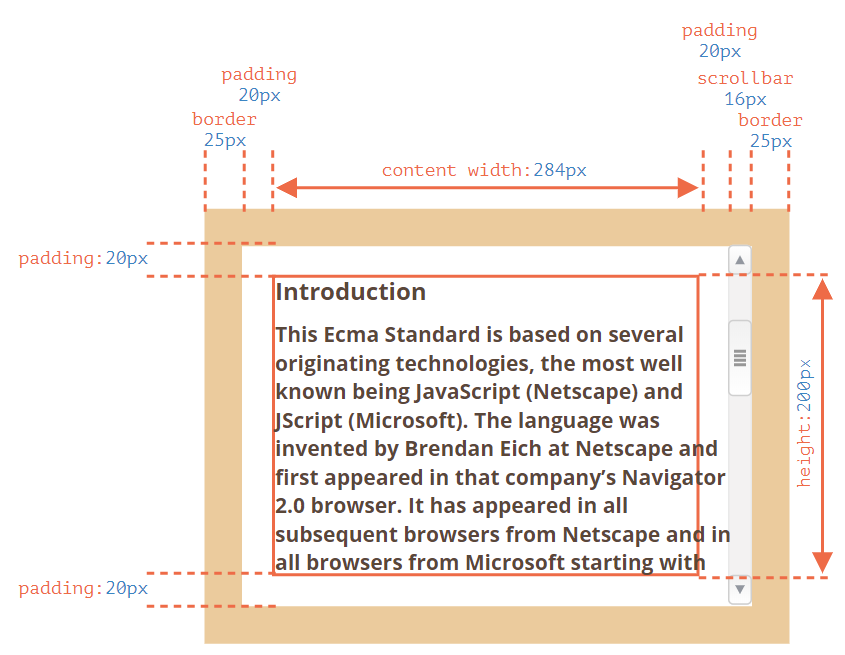
**overflow**: **auto**;

}

</**style**>

У элемента есть рамка (border), внутренний отступ (padding) и прокрутка. Полный набор характеристик. Обратите внимание, тут нет внешних отступов (margin), потому что они не являются частью элемента, для них нет особых JavaScript-свойств.

Результат выглядит так (Рисунок 5.7):



**Рисунок 5.7.** Свойства html-элемента

### Метрики

Общая картина с геометрическими свойствами представлениа на Рисунке 5.8.

Значениями свойств являются числа, подразумевается, что они в пикселях.

### offsetParent, offsetLeft/Top

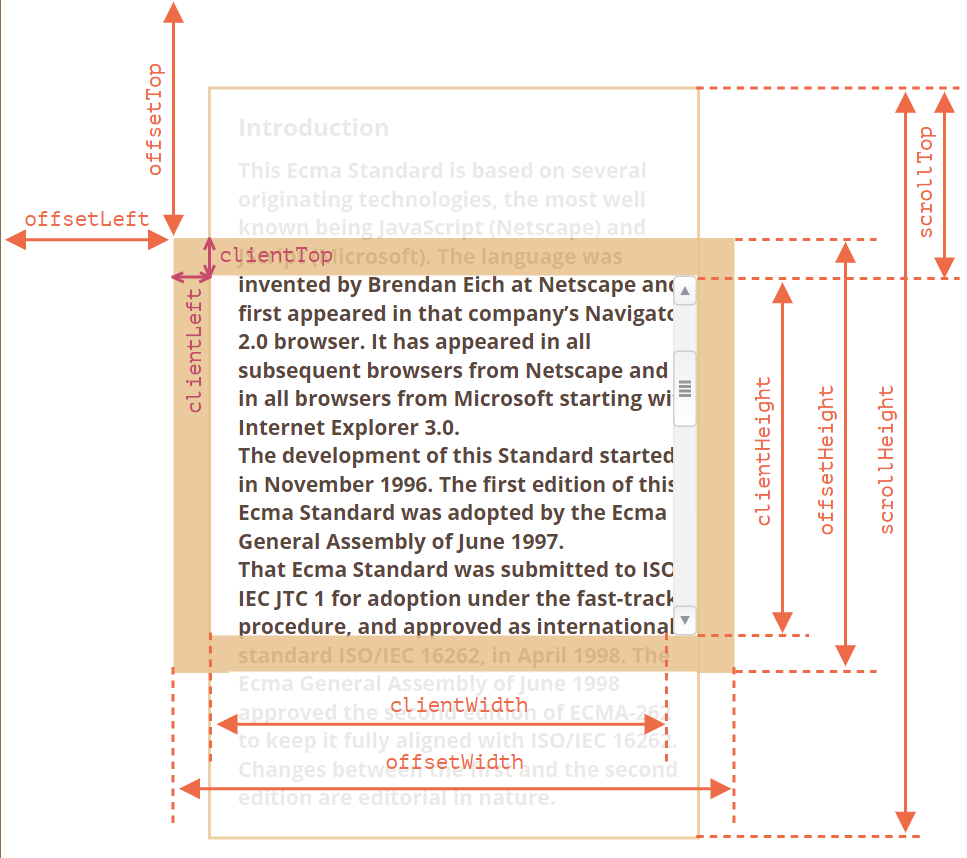
Эти свойства редко используются, но так как они являются «самыми внешними» метриками, мы начнём с них.

В свойстве offsetParent находится предок элемента, который используется внутри браузера для вычисления координат при рендеринге.

То есть, ближайший предок, который удовлетворяет следующим условиям:

* Является CSS-позиционированным (CSS-свойство position равно absolute, relative, fixed или sticky),
* или <td>, <th>, <table>,
* или <body>.

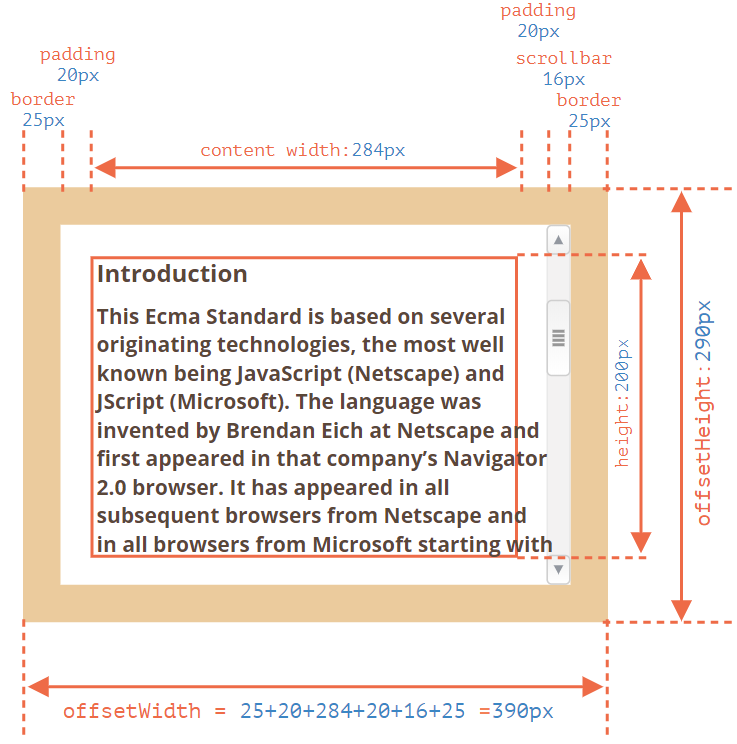
Свойства offsetLeft/offsetTop содержат координаты x/y относительно верхнего левого угла offsetParent.



**Рисунок 5.8.** Геометрические свойства html-элемента

### offsetWidth/Height

Эти два свойства – самые простые. Они содержат «внешнюю» ширину/высоту элемента, то есть его полный размер, включая рамки (Рисунок 5.9).

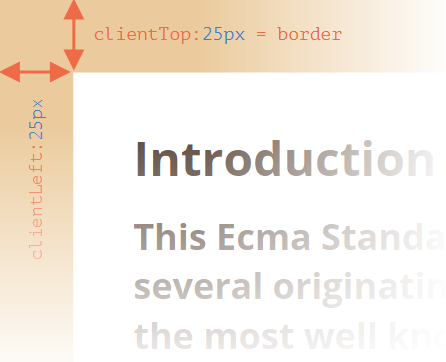


**Рисунок 5.9.** offsetWidth/Height

### clientTop/Left

Пойдём дальше. Внутри элемента у нас рамки (border).

Для них есть свойства-метрики clientTop и clientLeft (Рисунок 5.10).

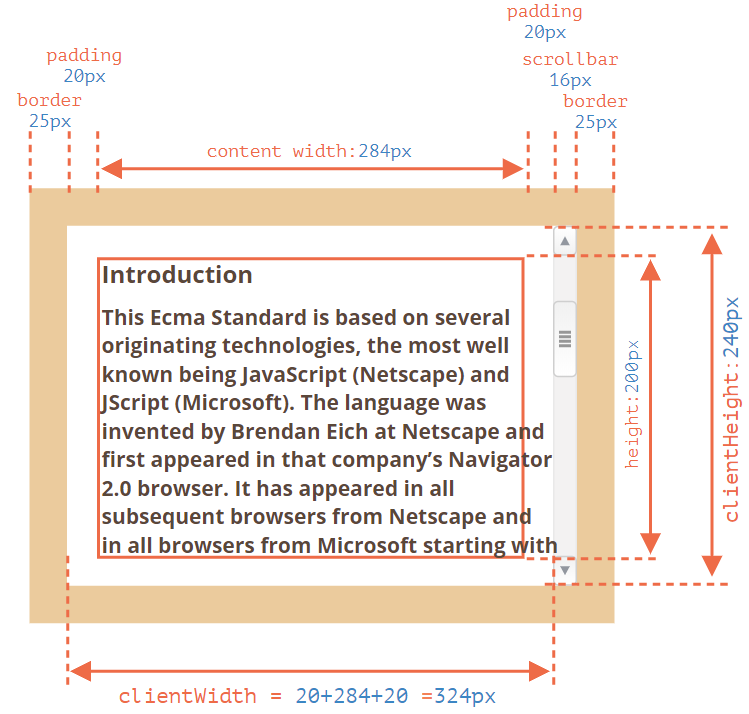


**Рисунок 5.10.** clientTop/Left

### clientWidth/Height

Эти свойства – размер области внутри рамок элемента.

Они включают в себя ширину области содержимого вместе с внутренними отступами padding, но без прокрутки (Рисунок 5.11).



**Рисунок 5.11.** clientWidth/Height

**Если нет внутренних отступов padding, то clientWidth/Height в точности равны размеру области содержимого внутри рамок и полосы прокрутки (если она есть).**

### scrollWidth/Height

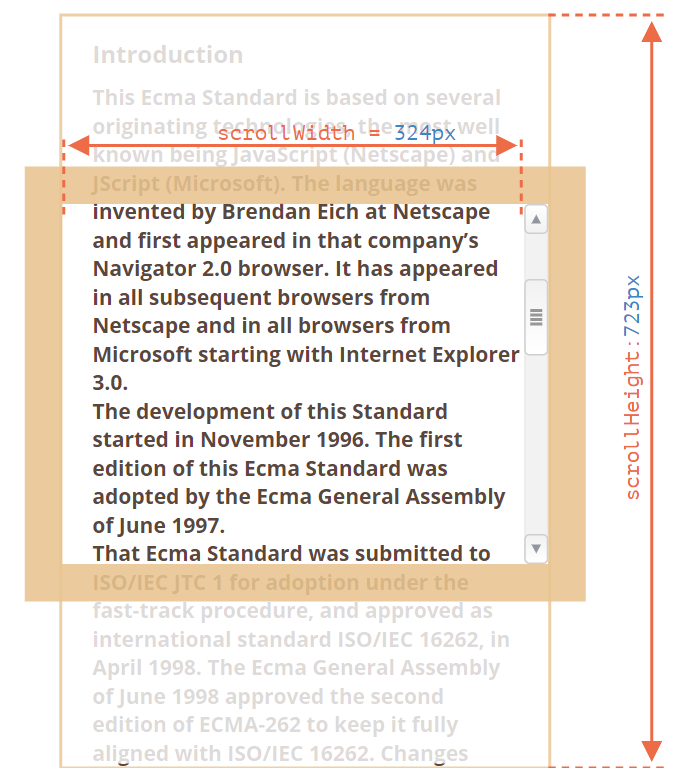
Эти свойства – как clientWidth/clientHeight, но также включают в себя прокрученную (которую не видно) часть элемента (Рисунок 5.12).

Эти свойства можно использовать, чтобы «распахнуть» элемент на всю ширину/высоту.

Таким кодом:

*// распахнуть элемент на всю высоту*

element.**style**.**height** = **`**${element.**scrollHeight**}**px`**;



**Рисунок 5.12.** scrollWidth/Height

### scrollLeft/scrollTop

Свойства scrollLeft/scrollTop – ширина/высота невидимой, прокрученной в данный момент, части элемента слева и сверху.

Другими словами, свойство scrollTop – это «сколько уже прокручено вверх».

В отличие от большинства свойств, которые доступны только для чтения, значения scrollLeft/scrollTop можно изменять, и браузер выполнит прокрутку элемента…

## Координаты

Чтобы передвигать элементы по экрану, нам следует познакомиться с системами координат.

Большинство соответствующих методов JavaScript работают в одной из двух указанных ниже систем координат:

1. **Относительно окна браузера** – как position:fixed, отсчёт идёт от верхнего левого угла окна.
   * мы будем обозначать эти координаты как clientX/clientY, причина выбора таких имён будет ясна позже, когда мы изучим свойства событий.
2. **Относительно документа** – как position:absolute на уровне документа, отсчёт идёт от верхнего левого угла документа.
   * мы будем обозначать эти координаты как pageX/pageY.

Когда страница полностью прокручена в самое начало, то верхний левый угол окна совпадает с левым верхним углом документа, при этом обе этих системы координат тоже совпадают. Но если происходит прокрутка, то координаты элементов в контексте окна меняются, так как они двигаются, но в то же время их координаты относительно документа остаются такими же.

### Координаты относительно окна: getBoundingClientRect

Метод elem.getBoundingClientRect() возвращает координаты в контексте окна для минимального по размеру прямоугольника, который заключает в себе элемент elem, в виде объекта встроенного класса DOMRect.

Основные свойства объекта типа DOMRect:

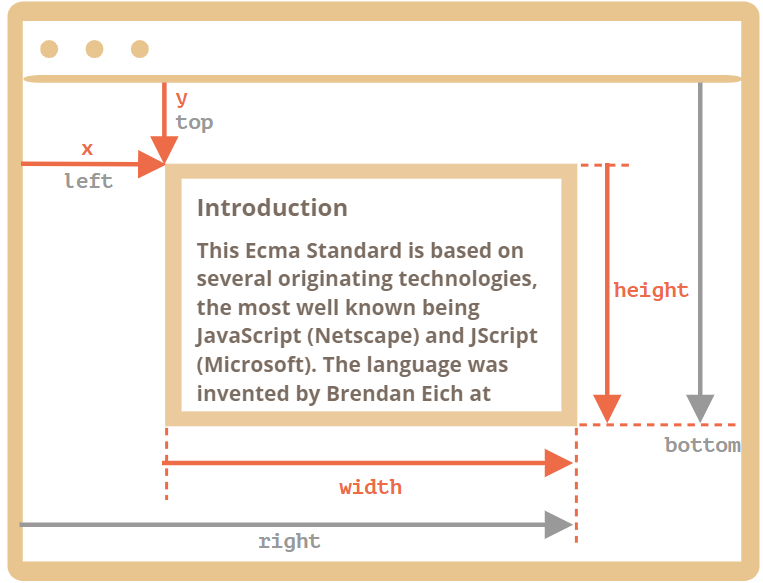
* x/y – X/Y-координаты начала прямоугольника относительно окна,
* width/height – ширина/высота прямоугольника (могут быть отрицательными).

Дополнительные, «зависимые», свойства:

* top/bottom – Y-координата верхней/нижней границы прямоугольника,
* left/right – X-координата левой/правой границы прямоугольника.

Изображение с результатами вызова elem.getBoundingClientRect() представлено на Рисунке 5.13.

Как вы видите, x/y и width/height уже точно задают прямоугольник. Остальные свойства могут быть легко вычислены на их основе



**Рисунок 5.13.** Координаты элемента относительно окна

### elementFromPoint(x, y)

Вызов document.elementFromPoint(x, y) возвращает самый глубоко вложенный элемент в окне, находящийся по координатам (x, y).

Синтаксис:

**let *elem*** = ***document***.elementFromPoint(x, y);

Например, код ниже выделяет с помощью стилей и выводит имя тега элемента, который сейчас в центре окна браузера (Листинг 5.25).

**Листинг 5.25** Использование elementFromPoint

**let *centerX*** = ***document***.**documentElement**.**clientWidth** / 2;

**let *centerY*** = ***document***.**documentElement**.**clientHeight** / 2;

**let *elem*** = ***document***.elementFromPoint(***centerX***, ***centerY***);

***elem***.**style**.**background** = **"red"**;

*alert*(***elem***.**tagName**);

Поскольку используются координаты в контексте окна, то элемент может быть разным, в зависимости от того, какая сейчас прокрутка.

### Координаты относительно документа

В такой системе координат отсчёт ведётся от левого верхнего угла документа, не окна.

В CSS координаты относительно окна браузера соответствуют свойству position:fixed, а координаты относительно документа – свойству position:absolute на самом верхнем уровне вложенности.

Мы можем воспользоваться свойствами position:absolute и top/left, чтобы привязать что-нибудь к конкретному месту в документе. При этом прокрутка страницы не имеет значения. Но сначала нужно получить верные координаты.

Не существует стандартного метода, который возвращал бы координаты элемента относительно документа, но мы можем написать его сами.

Две системы координат связаны следующими формулами:

* pageY = clientY + высота вертикально прокрученной части документа.
* pageX = clientX + ширина горизонтально прокрученной части документа.

Функция getCoords(elem) берёт координаты в контексте окна с помощью elem.getBoundingClientRect() и добавляет к ним значение соответствующей прокрутки (Листинг 5.26)

**Листинг 5.26** Вычисление координат относительно документа

*// получаем координаты элемента в контексте документа*

**function** *getCoords*(elem) {

**let** box = elem.getBoundingClientRect();

**return** {

**top**: box.**top** + ***pageYOffset***,

**left**: box.**left** + ***pageXOffset***

};

}

### Практическая работа №20

1. Создайте функцию positionAt(anchor, position, elem), которая позиционирует элемент elem в зависимости от значения свойства position рядом с элементом anchor.

Аргумент position – строка с одним из 3 значений:

"top" – расположить elem прямо над anchor

"right" – расположить elem непосредственно справа от anchor

"bottom" – расположить elem прямо под anchor

# 

# 6. Браузерные события

## Введение в браузерные события

*Событие* – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы (хотя события бывают и не только в DOM).

Вот список самых часто используемых DOM-событий, пока просто для ознакомления:

**События мыши:**

* click – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании).
* contextmenu – происходит, когда кликнули на элемент правой кнопкой мыши.
* mouseover / mouseout – когда мышь наводится на / покидает элемент.
* mousedown / mouseup – когда нажали / отжали кнопку мыши на элементе.
* mousemove – при движении мыши.

**События на элементах управления:**

* submit – пользователь отправил форму <form>.
* focus – пользователь фокусируется на элементе, например нажимает на <input>.

**Клавиатурные события:**

* keydown и keyup – когда пользователь нажимает / отпускает клавишу.

**События документа**:

* DOMContentLoaded – когда HTML загружен и обработан, DOM документа полностью построен и доступен.

**CSS events:**

* transitionend – когда CSS-анимация завершена.

### Обработчики событий

Событию можно назначить *обработчик*, то есть функцию, которая сработает, как только событие произошло.

Именно благодаря обработчикам JavaScript-код может реагировать на действия пользователя.

Есть несколько способов назначить событию обработчик. Сейчас мы их рассмотрим, начиная с самого простого.

### Использование атрибута HTML

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется on<событие>.

Например, чтобы назначить обработчик события click на элементе input, можно использовать атрибут onclick, вот так:

<**input value="Нажми меня" onclick="***alert*(**'Клик!'**)**" type="button"**>

При клике мышкой на кнопке выполнится код, указанный в атрибуте onclick.

Обратите внимание, для содержимого атрибута onclick используются одинарные кавычки, так как сам атрибут находится в двойных. Если мы забудем об этом и поставим двойные кавычки внутри атрибута, вот так: onclick="alert("Click!")", код не будет работать.

Атрибут HTML-тега – не самое удобное место для написания большого количества кода, поэтому лучше создать отдельную JavaScript-функцию и вызвать её там.

Следующий пример по клику запускает функцию countRabbits() (Листинг 6.1)

**Листинг 6.1** Обработчик события, назначенный через html

<**script**>

**function** *countRabbits*() {

**for**(**let** i=1; i<=3; i++) {

*alert*(**"Кролик номер "** + i);

}

}

</**script**>

<**input type="button" onclick="***countRabbits*()**" value="Считать кроликов!"**>

Как мы помним, атрибут HTML-тега не чувствителен к регистру, поэтому ONCLICK будет работать так же, как onClick и onCLICK… Но, как правило, атрибуты пишут в нижнем регистре: onclick.

### Использование свойства DOM-объекта

Можно назначать обработчик, используя свойство DOM-элемента on<событие>.

К примеру, elem.onclick (Листинг 6.2):

**Листинг 6.2** Обработчик события, назначенный через свойство DOM

<**input id="elem" type="button" value="Нажми меня!"**>

<**script**>

elem.onclick = **function**() {

*alert*(**'Спасибо'**);

};

</**script**>

Если обработчик задан через атрибут, то браузер читает HTML-разметку, создаёт новую функцию из содержимого атрибута и записывает в свойство.

Этот способ, по сути, аналогичен предыдущему.

**Обработчик всегда хранится в свойстве DOM-объекта, а атрибут – лишь один из способов его инициализации.**

**Так как у элемента DOM может быть только одно свойство с именем onclick, то назначить более одного обработчика так нельзя.**

### Доступ к элементу через this

Внутри обработчика события this ссылается на текущий элемент, то есть на тот, на котором, как говорят, «висит» (т.е. назначен) обработчик.

В коде ниже button выводит своё содержимое, используя this.innerHTML:

<**button onclick="***alert*(**this**.**innerHTML**)**"**>Нажми меня</**button**>

### addEventListener

Фундаментальный недостаток описанных выше способов назначения обработчика –- невозможность повесить несколько обработчиков на одно событие.

Например, одна часть кода хочет при клике на кнопку делать её подсвеченной, а другая – выдавать сообщение.

Разработчики стандартов достаточно давно это поняли и предложили альтернативный способ назначения обработчиков при помощи специальных методов addEventListener и removeEventListener. Они свободны от указанного недостатка.

Синтаксис добавления обработчика:

*element.addEventListener(event, handler[, options]);*

**event**

Имя события, например "click".

**handler**

Ссылка на функцию-обработчик.

**options**

Дополнительный объект со свойствами

Для удаления обработчика следует использовать removeEventListener:

*element.removeEventListener(event, handler[, options]);*

Метод addEventListener позволяет добавлять несколько обработчиков на одно событие одного элемента (Листинг 6.3).

**Листинг 6.3** Обработчик события, назначенный через addEventListener

<**input id="elem" type="button" value="Нажми меня"**/>

<**script**>

**function** *handler1*() {

*alert*(**'Спасибо!'**);

};

**function** *handler2*() {

*alert*(**'Спасибо ещё раз!'**);

}

elem.onclick = () => *alert*(**"Привет"**);

elem.addEventListener(**"click"**, *handler1*); *// Спасибо!*

elem.addEventListener(**"click"**, *handler2*); *// Спасибо ещё раз!*

</**script**>

### Объект события

Чтобы хорошо обработать событие, могут понадобиться детали того, что произошло. Не просто «клик» или «нажатие клавиши», а также – какие координаты указателя мыши, какая клавиша нажата и так далее.

Когда происходит событие, браузер создаёт *объект события*, записывает в него детали и передаёт его в качестве аргумента функции-обработчику.

Пример ниже демонстрирует получение координат мыши из объекта события (Листинг 6.4):

**Листинг 6.4** Объект события

<**input type="button" value="Нажми меня" id="elem"**>

<**script**>

elem.onclick = **function**(event) {

*// вывести тип события, элемент и координаты клика*

*alert*(event.**type** + **" на "** + event.**currentTarget**);

*alert*(**"Координаты: "** + event.**clientX** + **":"** + event.**clientY**);

};

</**script**>

Некоторые свойства объекта event:

**event.type**

Тип события, в данном случае "click".

**event.currentTarget**

Элемент, на котором сработал обработчик. Значение – обычно такое же, как и у this, но если обработчик является функцией-стрелкой или при помощи bind привязан другой объект в качестве this, то мы можем получить элемент из event.currentTarget.

**event.clientX / event.clientY**

Координаты курсора в момент клика относительно окна, для событий мыши.

### Действия браузера по умолчанию

Многие события автоматически влекут за собой действие браузера.

Например:

* Клик по ссылке инициирует переход на новый URL.
* Нажатие на кнопку «отправить» в форме – отсылку её на сервер.
* Зажатие кнопки мыши над текстом и её движение в таком состоянии – инициирует его выделение.

Если мы обрабатываем событие в JavaScript, то зачастую такое действие браузера нам не нужно. К счастью, его можно отменить.

#### Отмена действия браузера

Есть два способа отменить действие браузера:

* Основной способ – это воспользоваться объектом event. Для отмены действия браузера существует стандартный метод event.preventDefault().
* Если же обработчик назначен через on<событие> (не через addEventListener), то также можно вернуть false из обработчика.

В следующем примере при клике по ссылке переход не произойдёт:

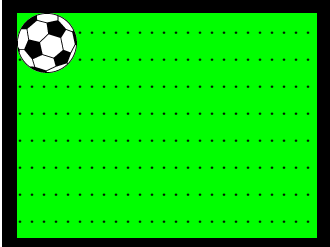
<**a href="/" onclick="return false"**>Нажми здесь</**a**>

или

<**a href="/" onclick="*event***.preventDefault()**"**>здесь</**a**>

### Практическая работа №21

1. Добавьте JavaScript к кнопке button, чтобы при нажатии элемент <div id="text"> исчезал.
2. Создайте кнопку, которая будет скрывать себя по нажатию.
3. Пусть мяч перемещается при клике на поле, туда, куда был клик



Требования:

* Центр мяча должен совпадать с местом нажатия мыши (если это возможно без пересечения краёв поля);
* CSS-анимация желательна, но не обязательна;
* Мяч ни в коем случае не должен пересекать границы поля;
* При прокрутке страницы ничего не должно ломаться;

Заметки:

* Код должен уметь работать с различными размерами мяча и поля, не привязываться к каким-либо фиксированным значениям.
* Используйте свойства event.clientX/event.clientY для определения координат мыши при клике.

1. Есть список сообщений.

При помощи JavaScript для каждого сообщения добавьте в верхний правый угол кнопку закрытия.

## Основы событий мыши

В этой главе мы более детально рассмотрим события мыши и их свойства.

Сразу заметим: эти события бывают не только из-за мыши, но и эмулируются на других устройствах, в частности, на мобильных, для совместимости.

### Типы событий мыши

Мы можем разделить события мыши на две категории: «простые» и «комплексные».

#### Простые события

Самые часто используемые простые события:

**mousedown/mouseup**

Кнопка мыши нажата/отпущена над элементом.

**mouseover/mouseout**

Курсор мыши появляется над элементом и уходит с него.

**mousemove**

Каждое движение мыши над элементом генерирует это событие.

**contextmenu**

Вызывается при попытке открытия контекстного меню, как правило, нажатием правой кнопки мыши. Но, заметим, это не совсем событие мыши, оно может вызываться и специальной клавишей клавиатуры.

#### Комплексные события

**click**

Вызывается при mousedown , а затем mouseup над одним и тем же элементом, если использовалась левая кнопка мыши.

**dblclick**

Вызывается двойным кликом на элементе.

Комплексные события состоят из простых, поэтому в теории мы могли бы без них обойтись. Но хорошо, что они существуют, потому что работать с ними очень удобно.

#### Порядок событий

Одно действие может вызвать несколько событий.

Например, клик мышью вначале вызывает mousedown, когда кнопка нажата, затем mouseup и click, когда она отпущена.

В случае, когда одно действие инициирует несколько событий, порядок их выполнения фиксирован. То есть обработчики событий вызываются в следующем порядке: mousedown → mouseup → click.

### Получение информации о кнопке: which

События, связанные с кликом, всегда имеют свойство which, которое позволяет определить нажатую кнопку мыши.

Это свойство не используется для событий click и contextmenu, поскольку первое происходит только при нажатии левой кнопкой мыши, а второе – правой.

Но если мы отслеживаем mousedown и mouseup, то оно нам нужно, потому что эти события срабатывают на любой кнопке, и which позволяет различать между собой «нажатие правой кнопки» и «нажатие левой кнопки».

Есть три возможных значения:

* event.which == 1 – левая кнопка
* event.which == 2 – средняя кнопка
* event.which == 3 – правая кнопка

Средняя кнопка сейчас – скорее экзотика, и используется очень редко.

### Модификаторы: shift, alt, ctrl и meta

Все события мыши включают в себя информацию о нажатых клавишах-модификаторах.

Свойства объекта события:

* shiftKey: Shift
* altKey: Alt (или Opt для Mac)
* ctrlKey: Ctrl
* metaKey: Cmd для Mac

Они равны true, если во время события была нажата соответствующая клавиша.

Например, кнопка внизу работает только при комбинации Alt+Shift+клик (Листинг 6.5).

**Листинг 6.5** Модификаторы

<**button id="button"**>Нажми Alt+Shift+Click на мне!</**button**>

<**script**>

button.onclick = **function**(event) {

**if** (event.**altKey** && event.**shiftKey**) {

*alert*(**'Ура!'**);

}

};

</**script**>

### Координаты: clientX/Y, pageX/Y

Все события мыши имеют координаты двух видов:

1. Относительно окна: clientX и clientY.
2. Относительно документа: pageX и pageY.

### Практическая работа №22

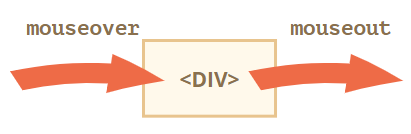
1. Создайте список, в котором элементы могут быть выделены, как в файловых менеджерах.

* При клике на элемент списка выделяется только этот элемент (добавляется класс .selected), отменяется выделение остальных элементов.
* Если клик сделан вместе с Ctrl (Cmd для Mac), то выделение переключается на элементе, но остальные элементы при этом не изменяются.

## Движение мыши

### События mouseover/mouseout, relatedTarget

Событие mouseover происходит в момент, когда курсор оказывается над элементом, а событие mouseout – в момент, когда курсор уходит с элемента (Рисунок 6.1).



**Рисунок 6.1.** mouseover/mouseout

Эти события являются особенными, потому что у них имеется свойство relatedTarget. Оно «дополняет» target. Когда мышь переходит с одного элемента на другой, то один из них будет target, а другой relatedTarget.

Для события mouseover:

* event.target – это элемент, на который курсор перешёл.
* event.relatedTarget – это элемент, с которого курсор ушёл (relatedTarget → target).

Для события mouseout наоборот:

* event.target – это элемент, с которого курсор ушёл.
* event.relatedTarget – это элемент, на который курсор перешёл (target → relatedTarget).

### Пропуск элементов

Событие mousemove происходит при движении мыши. Однако, это не означает, что указанное событие генерируется при прохождении каждого пикселя.

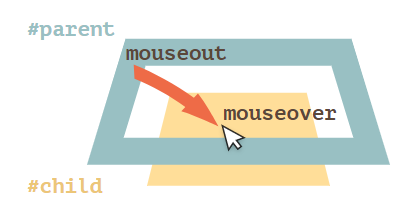
Браузер периодически проверяет позицию курсора и, заметив изменения, генерирует события mousemove.

Это означает, что если пользователь двигает мышкой очень быстро, то некоторые DOM-элементы могут быть пропущены:

### Событие mouseout при переходе на потомка

Важная особенность события mouseout – оно генерируется в том числе, когда указатель переходит с элемента на его потомка.

То есть, визуально указатель всё ещё на элементе, но мы получим mouseout (Рисунок 6.2)!



**Рисунок 6.2.** mouseover при переходе на потомка

Это выглядит странно, но легко объясняется.

**По логике браузера, курсор мыши может быть только над одним элементом в любой момент времени – над самым глубоко вложенным и верхним по z-index.**

Таким образом, если курсор переходит на другой элемент (пусть даже дочерний), то он покидает предыдущий.

### События mouseenter и mouseleave

События mouseenter/mouseleave похожи на mouseover/mouseout. Они тоже генерируются, когда курсор мыши переходит на элемент или покидает его.

Но есть и пара важных отличий:

1. Переходы внутри элемента, на его потомки и с них, не считаются.
2. События mouseenter/mouseleave не всплывают.

События mouseenter/mouseleave предельно просты и понятны.

Когда указатель появляется над элементом – генерируется mouseenter, причём не имеет значения, где именно указатель: на самом элементе или на его потомке.

Событие mouseleave происходит, когда курсор покидает элемент.

## Drag'n'Drop с событиями мыши

Drag’n’Drop – отличный способ улучшить интерфейс. Захват элемента мышкой и его перенос визуально упростят что угодно: от копирования и перемещения документов (как в файловых менеджерах) до оформления заказа («положить в корзину»).

В современном стандарте HTML5 есть раздел о Drag and Drop – и там есть специальные события именно для Drag’n’Drop переноса, такие как dragstart, dragend и так далее.

Они интересны тем, что позволяют легко решать простые задачи. Например, можно перетащить файл в браузер, так что JS получит доступ к его содержимому.

Но у них есть и ограничения. Например, нельзя организовать перенос «только по горизонтали» или «только по вертикали». Также нельзя ограничить перенос внутри заданной зоны. Есть и другие интерфейсные задачи, которые такими встроенными событиями не реализуемы. Кроме того, мобильные устройства плохо их поддерживают.

Здесь мы будем рассматривать Drag’n’Drop при помощи событий мыши.

### Алгоритм Drag’n’Drop

Базовый алгоритм Drag’n’Drop выглядит так:

1. При mousedown – готовим элемент к перемещению, если необходимо (например, создаём его копию).
2. Затем при mousemove передвигаем элемент на новые координаты путём смены left/top и position:absolute.
3. При mouseup – остановить перенос элемента и произвести все действия, связанные с окончанием Drag’n’Drop.

В следующем примере эти шаги реализованы для переноса мяча (Листинг 6.6).

**Листинг 6.6** Перенос мяча

ball.onmousedown = **function**(event) { *// (1) отследить нажатие*

*// (2) подготовить к перемещению:*

*// разместить поверх остального содержимого и в абсолютных координатах*

ball.**style**.**position** = **'absolute'**;

ball.**style**.**zIndex** = 1000;

*// переместим в body, чтобы мяч был точно не внутри position:relative*

***document***.**body**.append(ball);

*// и установим абсолютно спозиционированный мяч под курсор*

moveAt(event.**pageX**, event.**pageY**);

*// передвинуть мяч под координаты курсора*

*// и сдвинуть на половину ширины/высоты для центрирования*

**function** moveAt(pageX, pageY) {

ball.**style**.**left** = pageX - ball.**offsetWidth** / 2 + **'px'**;

ball.**style**.**top** = pageY - ball.**offsetHeight** / 2 + **'px'**;

}

**function** onMouseMove(event) {

moveAt(event.**pageX**, event.**pageY**);

}

*// (3) перемещать по экрану*

***document***.addEventListener(**'mousemove'**, onMouseMove);

*// (4) положить мяч, удалить более ненужные обработчики событий*

ball.onmouseup = **function**() {

***document***.removeEventListener(**'mousemove'**, onMouseMove);

ball.**onmouseup** = **null**;

};

};

Если запустить этот код, то мы заметим нечто странное. При начале переноса мяч «раздваивается» и переносится не сам мяч, а его «клон».

Всё потому, что браузер имеет свой собственный Drag’n’Drop, который автоматически запускается и вступает в конфликт с нашим. Это происходит именно для картинок и некоторых других элементов.

Его нужно отключить:

ball.ondragstart = **function**() {

**return false**;

};

### Цели переноса (droppable)

В предыдущих примерах мяч можно было бросить просто где угодно в пределах окна. В реальности мы обычно берём один элемент и перетаскиваем в другой. Например, «файл» в «папку» или что-то ещё.

Абстрактно говоря, мы берём перетаскиваемый (draggable) элемент и помещаем его в другой элемент «цель переноса» (droppable).

Нам нужно знать:

* куда пользователь положил элемент в конце переноса, чтобы обработать его окончание
* и, желательно, над какой потенциальной целью (элемент, куда можно положить, например, изображение папки) он находится в процессе переноса, чтобы подсветить её.

Решение довольно интересное и немного хитрое, давайте рассмотрим его.

Какой может быть первая мысль? Возможно, установить обработчики событий mouseover/mouseup на элемент – потенциальную цель переноса?

Но это не работает.

Проблема в том, что при перемещении перетаскиваемый элемент всегда находится поверх других элементов. А события мыши срабатывают только на верхнем элементе, но не на нижнем.

Существует метод document.elementFromPoint(clientX, clientY). Он возвращает наиболее глубоко вложенный элемент по заданным координатам окна (или null, если указанные координаты находятся за пределами окна).

Мы можем использовать его, чтобы из любого обработчика событий мыши выяснить, над какой мы потенциальной целью переноса, вот так (Листинг 6.7):

**Листинг 6.7** Получение элемента под курсором

*// внутри обработчика события мыши*

ball.**hidden** = **true**; *// (\*) прячем переносимый элемент*

**let *elemBelow*** = ***document***.elementFromPoint(***event***.**clientX**, ***event***.**clientY**);

*// elemBelow - элемент под мячом (возможная цель переноса)*

ball.**hidden** = **false**;

Заметим, нам нужно спрятать мяч перед вызовом функции (\*). В противном случае по этим координатам мы будем получать мяч, ведь это и есть элемент непосредственно под указателем: elemBelow=ball. Так что мы прячем его и тут же показываем обратно.

Мы можем использовать этот код для проверки того, над каким элементом мы «летим», в любое время. И обработать окончание переноса, когда оно случится.

Расширенный код onMouseMove с поиском потенциальных целей переноса (Листинг 6.8):

**Листинг 6.8** Расширенный код onMouseMove

*// потенциальная цель переноса, над которой мы пролетаем прямо сейчас*

**let *currentDroppable*** = **null**;

**function** *onMouseMove*(event) {

moveAt(event.**pageX**, event.**pageY**);

ball.**hidden** = **true**;

**let** elemBelow = ***document***.elementFromPoint(event.**clientX**, event.**clientY**);

ball.**hidden** = **false**;

*// событие mousemove может произойти и когда указатель за пределами окна*

*// (мяч перетащили за пределы экрана)*

*// если clientX/clientY за пределами окна, elementFromPoint вернёт null*

**if** (!elemBelow) **return**;

*// потенциальные цели переноса помечены классом droppable (может быть и другая логика)*

**let** droppableBelow = elemBelow.closest(**'.droppable'**);

**if** (***currentDroppable*** != droppableBelow) {

*// мы либо залетаем на цель, либо улетаем из неё*

*// внимание: оба значения могут быть null*

*// currentDroppable=null,*

*// если мы были не над droppable до этого события (например, над пустым пространством)*

*// droppableBelow=null,*

*// если мы не над droppable именно сейчас, во время этого события*

**if** (***currentDroppable***) {

*// логика обработки процесса "вылета" из droppable (удаляем подсветку)*

leaveDroppable(***currentDroppable***);

}

***currentDroppable*** = droppableBelow;

**if** (***currentDroppable***) {

*// логика обработки процесса, когда мы "влетаем" в элемент droppable*

enterDroppable(***currentDroppable***);

}

}

}

### Практическая работа №23

1. Создайте слайдер:



Важные детали:

* Слайдер должен нормально работать при резком движении мыши влево или вправо за пределы полосы. При этом бегунок должен останавливаться чётко в нужном конце полосы.
* При нажатом бегунке мышь может выходить за пределы полосы слайдера, но слайдер пусть все равно работает (это удобно для пользователя).

## Клавиатура: keydown и keyup

События клавиатуры должны использоваться, если мы хотим обрабатывать взаимодействие пользователя с клавиатурой (в том числе виртуальной). К примеру, если нам нужно реагировать на стрелочные клавиши Up и Down или горячие клавиши (включая комбинации клавиш).

### События keydown и keyup

Событие keydown происходит при нажатии клавиши, а keyup – при отпускании.

Свойство key объекта события позволяет получить символ, а свойство code – «физический код клавиши».

К примеру, одну и ту же клавишу Z можно нажать с клавишей Shift и без неё. В результе получится два разных символа: z в нижнем регистре и Z в верхнем регистре.

Свойство event.key – это непосредственно символ, и он может различаться. Но event.code всегда будет тот же

Если пользователь работает с разными языками, то при переключении на другой язык символ изменится с "Z" на совершенно другой. Получившееся станет новым значением event.key, тогда как event.code останется тем же: "KeyZ".

### Автоповтор

При долгом нажатии клавиши возникает автоповтор: keydown срабатывает снова и снова, и когда клавишу отпускают, то отрабатывает keyup. Так что ситуация, когда много keydownи один keyup, абсолютно нормальна.

Для событий, вызванных автоповтором, у объекта события свойство event.repeat равно true.

### Действия по умолчанию

Действия по умолчанию весьма разнообразны, много чего можно инициировать нажатием на клавиатуре.

Для примера:

* Появление символа (самое очевидное).
* Удаление символа (клавиша Delete).
* Прокрутка страницы (клавиша PageDown).
* Открытие диалогового окна браузера «Сохранить» (Ctrl+S)
* …и так далее.

Предотвращение стандартного действия с помощью event.preventDefault() работает практически во всех сценариях, кроме тех, которые происходят на уровне операционной системы. Например, комбинация Alt+F4 инициирует закрытие браузера в Windows, что бы мы ни делали в JavaScript.

Для примера, <input> ниже ожидает телефонный номер, так что ничего кроме чисел, +, () или - принято не будет (Листинг 6.9):

**Листинг 6.9** Пример предотвращения стандартного действия

<**script**>

**function** *checkPhoneKey*(key) {

**return** (key >= **'0'** && key <= **'9'**) || key == **'+'** || key == **'('** || key == **')'** || key == **'-'** ||

key == **'ArrowLeft'** || key == **'ArrowRight'** || key == **'Delete'** || key == **'Backspace'**;

}

</**script**>

<**input onkeydown="return** *checkPhoneKey*(***event***.**key**)**" placeholder="Введите телефон" type="tel"**>

## Проверочная работа №3

Выполните предложенное ниже задание согласно своему варианту, составьте **отчет** с **блок-схемами** или **словесным** описанием всех алгоритмов, **кодом** и **результатами работы**.

### Варианты

1. Разработайте игру “Сапер”. Идея игры такая: на игровом поле где-то спрятаны мины. Игрок кликает по клеткам, открывая их. Если в клетке была мина, игрок проиграл. Если нет, то в клетке выводится цифра, показывающая общее число мин в соседних 8 клетках. Если игрок открыл все клетки, кроме заминированных, он победил. Если игрок открывает клетку, рядом с которой нет мин, то все соседние клетки открываются автоматически (если на них тоже нет мин, то процесс продолжается).

Правой кнопкой мыши на неоткрытых клетках можно расставлять флажки.

Надпись «Вы победили» или «Вы проиграли» должна выводиться в окошке поверх игрового поля и содержать кнопку «Новая игра».

При реализации игры необходимо предусмотреть три варианта экрана:

* стартовый экран - содержит наименование игры, поле ввода своего имени, а также кнопку “Начать игру”. Кнопка “Начать игру” активна, если введено имя пользователя.
* экран игры - интерфейс должен содержать:
  + блок с таймером - содержит таймер обратного отсчета с указанием минут, секунд в формате мм:сс.
  + блок с заработанными баллами (количество оставшихся секунд по таймеру) - количество заработанных игроком баллов за текущую игру.
  + блок с руководством по игре.
* экран окончания игры - содержит результат игрока и кнопку, позволяющую начать заново.

1. Вам необходимо разработать онлайн игру (аркаду) – «Гонки». Участнику необходимо реализовать ограниченное по размерам игровое поле, на котором в нижней части передвигается машина. В верхней части поля располагаются объекты, которые являются препятствиями или бонусами в игре. Скорость перемещения объектов увеличивается со временем игры. Задача игрока – продержаться в игре как можно больше времени.

**Стартовый экран** - содержит наименование игры, поле ввода своего имени, а также кнопку “Начать игру”. Имя не может быть пустым. Кнопка “Начать игру” активна, если только введено имя пользователя.

**Экран игры** - интерфейс должен содержать:

* блок с таймером - содержит таймер обратного отсчета с указанием минут, секунд в формате мм: сс. Таймер должен обновляться без перезагрузки страницы. В этом же блоке располагается кнопка “пауза”
* блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
* блок с именем игрока - если игрок на стартовом экране в имени ввел “terter” то имя пользователя должно отображаться по другому, так как это “кодовое” слово для игры в тестовом режиме.
* игровое поле - в зависимости от выбранной экспертами игры. Процесс игры должен включать в себя взаимодействия пользователя и компьютера и протекать без обновления страницы

**Экран окончания игры** - содержит результат игрока.

Реализуйте логику, в состав которой должны быть включены следующие функции:

* Пауза игрового процесса - останавливается время на таймере, запрещается воздействие на игровое поле, перечень элементов, для которых может сохранять анимацию, описывается экспертами отдельно (например, перемещение фона в режиме паузы). Режим паузы также может быть инициирован по нажатию на клавишу пробел. Возобновление игры так же возможно по нажатию на клавишу “пробел” или по нажатию на кнопку “Пауза” на игровом поле
* Таймер обратного отсчета - начинает обратный отсчет с началом игры, как только доходит до значения 00:00 игра заканчивается
* Режима теста - режим в котором таймер обратного отсчета не запускается, и игра не останавливается при взаимодействиях, которые подразумевают проигрыш или конец игры
* Реализация логики работы игрового поля, например, изменение количества собранных объектов, уменьшение количества “жизней”, уменьшение оставшегося времени и т.д., а также обработка соответствующих событий - конец игры, проигрыш, выигрыш при наступлении определенных действий.

1. С помощью чистого js реализуйте игру “Рыбки”.

**Общее описание игры.**

Существует аквариум, в котором плавают рыбки. Каждую секунду добавляются новые рыбки, до определенного максимума заполняемости аквариума. При клике на рыбку, она исчезает, игроку начисляются некоторое количество баллов. Задача, набрать максимальное количество баллов за минуту.

**Минимальный функционал программы:**

* ввод имени игроков перед началом игры
* по клику на рыбку она исчезает, а игроку начисляются очки
* рыбки двигаются
* игра заканчивается с окончанием времени

**Дополнительный функционал программы:**

* присутствуют несколько видов рыб разных размеров и видов, при клике на каждую начисляется разное количество очков
* присутствуют анимации появления и исчезновения рыб, анимации таймера, элементов интерфейса
* после окончания игры появляется таблица рекордов, в которой показан текущий результат игрока и результаты лучших 10 попыток.

1. С помощью чистого js реализуйте проект “Todo - список дел”. Составьте отчет по выполненной работе.

“Todo” очень простое в использовании приложение, благодаря которому можно составлять списки дел (заметок), зачеркивать готовые и упорядочивать их посредством блокнотов, которые можно также создать для конкретных событий или дел.

Под свой вкус можно выбрать шрифт и размер текста. На главной странице будут располагаться наши блокноты с указанием времени их создания. Новые блокноты можно создать, нажав на «плюсик» в верхнем углу экрана. В открывшемся окне сразу же нам нужно написать заголовок будущего блокнота, а после этого начать записывать свои дела.

**Минимальный функционал программы:**

* присутствует разработанный дизайн и адаптивная верстка
* можно создавать блокноты и списки дел в них
* можно отмечать выполненные дела, удалять и редактировать заметки и блокноты
* все данные сохраняются после обновления страницы

**Дополнительный функционал программы:**

* можно окрашивать блокноты в разные цвета
* можно осуществлять поиск по заметкам
* устанавливать приоритет дел в списке
* сортировать дела по приоритету, названию, дате
* присутствует анимация интерфейса

1. Требуется разработать игру «Space pipes».

**Сюжет:**

*При столкновении с космическим телом на Международной космической станции «Марс-2» в секторе «А» произошел процесс неконтролируемой разгерметизации.*

*Требуется срочно восстановить разрушенную систему подачи воздуха. Для этого в сектор «А» были направлены специалисты по монтажу систем вентиляции. Перед ними поставлена задача, восстановить систему воздуховода на поврежденном участке, соединив нужные участки труб между началом и концом разрыва (начало и конец разрыва находятся на противоположных сторонах).*

**Общие требования:**

Начало и конец трубы находятся в верхнем левом и нижнем правом углах соответственно. Клетки поля заполняются случайным образом изображениями фрагментов трубы. При каждом щелчке игрока по изображению фрагмент поворачивается на 90 градусов Игра заканчивается либо с построением правильного пути, либо с окончанием времени таймера.

При реализации игры необходимо предусмотреть три варианта экрана:

* стартовый экран - содержит наименование игры, поле ввода своего имени, а также кнопку “Начать игру”. Кнопка “Начать игру” активна, если введено имя пользователя.
* экран игры - интерфейс должен содержать:
  + блок с таймером - содержит таймер обратного отсчета с указанием минут, секунд в формате мм:сс.
  + блок с заработанными баллами (количество оставшихся секунд по таймеру) - количество заработанных игроком баллов за текущую игру.
  + блок с руководством по игре.
* экран окончания игры - содержит результат игрока и кнопку, позволяющую начать заново.

1. Разработайте электронный вариант игры “Быки и коровы”.

“Быки и коровы” — логическая игра, в ходе которой за несколько попыток один из игроков должен определить, что задумал другой игрок.

Каждый из игроков задумывает и записывает тайное 4-значное число с неповторяющимися цифрами. Игрок, который начинает игру по жребию, делает первую попытку отгадать число. Попытка — это 4-значное число с неповторяющимися цифрами, сообщаемое противнику. Противник сообщает в ответ, сколько цифр угадано без совпадения с их позициями в тайном числе (то есть количество коров) и сколько угадано вплоть до позиции в тайном числе (то есть количество быков). Например:

Задумано тайное число «3219».

Попытка: «2310».

Результат: две «коровы» (две цифры: «2» и «3» — угаданы на неверных позициях) и один «бык» (одна цифра «1» угадана вплоть до позиции).

Игроки делают попытки угадать по очереди. Побеждает тот, кто угадает число первым, при условии, что он не начинал игру. Если же отгадавший начинал игру — его противнику предоставляется последний шанс угадать последовательность.

При игре против компьютера игрок вводит комбинации одну за другой, пока не отгадает всю последовательность.

При реализации игры необходимо предусмотреть три варианта экрана:

* стартовый экран - содержит наименование игры, поле ввода своего имени, а также кнопку “Начать игру”. Кнопка “Начать игру” активна, если введено имя пользователя.
* экран игры - интерфейс должен содержать:
  + блок с таймером - содержит таймер обратного отсчета с указанием минут, секунд в формате мм:сс.
  + блок с заработанными баллами (количество оставшихся секунд по таймеру) - количество заработанных игроком баллов за текущую игру.
  + блок с руководством по игре.
* экран окончания игры - содержит результат игрока и кнопку, позволяющую начать заново.

1. К вам обратилась компания по разработке игр для веб-сайтов. Компания

просит помочь в разработке веб-игры.

**Описание игры:**

При открытии игры должен быть отображен **экран входа в игру**. На

данном экране присутствует поле для указания вашего имени и кнопка входа.

Если имя пользователя не указано, то кнопка должна быть не активна.

Когда имя пользователя будет заполнено кнопка должна стать активной.

При клике на кнопку входа экран должен смениться на игровой.

На **игровом экране** находиться имя пользователя, которое должно

корректно отображаться, секундомер (начинает считать с 00:00), который

должен быть запущен при старте игры, счетчик жизней (при старте у игрока

есть 3 жизни) и счетчик пойманных предметов.

На игровом поле есть корзина, в которую нужно ловить падающие

фрукты.

При старте игры корзина должна находиться снизу по центру.

В процессе игры должны появляться фрукты (1 фрукт каждую секунду).

Фрукты должны падать вниз с разной скоростью.

Игрок должен иметь возможность управлять корзиной с помощью

стрелок влево и вправо.

Корзина не должна выходить за пределы игровой зоны.

Если фрукт достигает земли (нижняя граница экрана), то у пользователя

отнимаются жизни.

Если удерживать «пробел», то все фрукты, присутствующие на экране

будут считаться собранными. Этот процесс может быть активен не дольше 4

секунд. После этого нужно 5 секунд на восстановление.

Постарайтесь отобразить процесс «автоматического сбора».

Должна быть возможность поставить игру на паузу нажав кнопку ESC.

Повторное нажатие должно продолжить игру. Во время паузы все

интерактивные действия (анимация, секундомер, фрукты, корзина) должны

быть приостановлены.

Когда жизни будут равны нулю, то игра должна закончиться и должен

отобразиться экран с результатами.

На **экране с результатами** необходимо отобразить время, которое игрок

продержался в игре и кол-во собранных фруктов. Если игрок продержался

дольше 10 секунд, то должно отобразиться сообщение о выигрыше, иначе о

проигрыше.

На экране с результатами есть кнопка «Играть сначала» при клике на

которую игра должна перезапуститься.

1. Вам необходимо разработать онлайн игру Арканоид. Цель игры – очистить поле от прямоугольных блоков, попадая по ним шариком, который отскакивает от «ракетки», который можно передвигать вправо и влево. Если шарик «упал на пол», уменьшается количество жизней. Когда жизни заканчиваются, заканчивается и игра..

**Стартовый экран** - содержит наименование игры, поле ввода своего имени, а также кнопку “Начать игру”. Имя не может быть пустым. Кнопка “Начать игру” активна, если только введено имя пользователя.

**Экран игры** - интерфейс должен содержать:

* блок с таймером - содержит таймер обратного отсчета с указанием минут, секунд в формате мм:сс. Таймер должен обновляться без перезагрузки страницы. В этом же блоке располагается кнопка “пауза”
* блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
* блок с именем игрока - если игрок на стартовом экране в имени ввел “terter” то имя пользователя должно отображаться по другому, так как это “кодовое” слово для игры в тестовом режиме.
* игровое поле - в зависимости от выбранной экспертами игры. Процесс игры должен включать в себя взаимодействия пользователя и компьютера и протекать без обновления страницы

**Экран окончания игры** - содержит таблицу результатов с первыми 10 лучшими результатами. Если игрок по баллам не вошел в 10 лучших, то вместо 10 места показывается его результат с указанием места в таблице рекордов.

Реализуйте логику, в состав которой должны быть включены следующие функции:

* Пауза игрового процесса - останавливается время на таймере, запрещается воздействие на игровое поле, перечень элементов, для которых может сохранять анимация описывается экспертами отдельно (например, перемещение фона в режиме паузы). Режим паузы также может быть инициирован по нажатию на клавишу пробел. Возобновление игры так же возможно по нажатию на клавишу “пробел” или по нажатию на кнопку “Пауза” на игровом поле
* Таймер обратного отсчета - начинает обратный отсчет с началом игры, как только доходит до значения 00:00 игра заканчивается
* Режима теста - режим в котором таймер обратного отсчета не запускается, и игра не останавливается при взаимодействиях, которые подразумевают проигрыш или конец игры
* Реализация логики работы игрового поля, например, изменение количества собранных объектов, уменьшение количества “жизней”, уменьшение оставшегося времени и т.д., а также обработка соответствующих событий - конец игры, проигрыш, выигрыш при наступлении определенных действий.

1. Вам необходимо разработать онлайн игру “Сбей самолет”. По экрану летят вражеские самолеты. Цель играющего – сбить их. Пусковая установка находится в нижней строке экрана. Пусковую установку можно перемещать по строке вперед и назад. Вражеские самолеты при полете сбрасывают бомбы, при попадании которых в пусковую установку ей наносится ущерб (уменьшается "здоровье" установки). Необходимо сбить максимальное количество самолетов за выделенное время.

**Стартовый экран** - содержит наименование игры, поле ввода своего имени, а также кнопку «Начать игру». Имя не может быть пустым. Кнопка «Начать игру» активна, если только введено имя пользователя.

**Экран игры** - интерфейс должен содержать:

* блок с таймером - содержит таймер обратного отсчета с указанием минут, секунд в формате мм:сс. Таймер должен обновляться без перезагрузки страницы. В этом же блоке располагается кнопка «пауза»
* блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
* блок с именем игрока - если игрок на стартовом экране в имени ввел «tester» то имя пользователя должно отображаться по другому, так как это «кодовое» слово для игры в тестовом режиме.
* игровое поле - в соответствии с описанием игры. Процесс игры должен включать в себя взаимодействия пользователя и компьютера и протекать без обновления страницы

**Экран окончания игры** - содержит результат игрока.

Реализуйте логику, в состав которой должны быть включены следующие функции:

* Пауза игрового процесса - останавливается время на таймере, запрещается воздействие на игровое поле. Режим паузы также может быть инициирован по нажатию на клавишу пробел. Возобновление игры так же возможно по нажатию на клавишу «пробел» или по нажатию на кнопку «Пауза» на игровом поле
* Таймер обратного отсчета - начинает обратный отсчет с началом игры, как только доходит до значения 00:00 игра заканчивается
* Сохранение итоговых результатов в базу данных на стороне сервера. Участнику предоставляется готовая структура базы данных(дамп таблицы), с которой он должен работать. Изменять структуру нельзя.
* Режима теста - режим в котором таймер обратного отсчета не запускается, и игра не останавливается при взаимодействиях, которые подразумевают проигрыш или конец игры
* Реализация логики работы игрового поля, например, изменение количества сбитых самолетов, уменьшение количества «здоровья пусковой установки», уменьшение оставшегося времени и т.д., а также обработка соответствующих событий - конец игры, проигрыш, выигрыш при наступлении определенных действий.

1. Вам необходимо разработать онлайн игру «StarTrek».

Цель игры – провести космический корабль через поле метеоритов. Если метеорит попал в корабль - уменьшается количество жизней. Когда жизни заканчиваются, заканчивается и игра.

**Стартовый экран** - содержит наименование игры, поле ввода своего имени, а также кнопку “Начать игру”. Имя не может быть пустым. Кнопка “Начать игру” активна, если только введено имя пользователя.

**Экран игры** - интерфейс должен содержать:

* блок с таймером - содержит таймер обратного отсчета с указанием минут, секунд в формате мм:сс. Таймер должен обновляться без перезагрузки страницы. В этом же блоке располагается кнопка “пауза”
* блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
* блок с именем игрока - если игрок на стартовом экране в имени ввел “tester” то имя пользователя должно отображаться по другому, так как это “кодовое” слово для игры в тестовом режиме.
* игровое поле - в зависимости от выбранной экспертами игры. Процесс игры должен включать в себя взаимодействия пользователя и компьютера и протекать без обновления страницы

**Экран окончания игры** - содержит результат игрока.

Реализуйте логику, в состав которой должны быть включены следующие функции:

* Пауза игрового процесса - останавливается время на таймере, запрещается воздействие на игровое поле, перечень элементов, для которых может сохранять анимацию, описывается экспертами отдельно (например, перемещение фона в режиме паузы). Режим паузы также может быть инициирован по нажатию на клавишу пробел. Возобновление игры так же возможно по нажатию на клавишу “пробел” или по нажатию на кнопку “Пауза” на игровом поле
* Таймер обратного отсчета - начинает обратный отсчет с началом игры, как только доходит до значения 00:00 игра заканчивается
* Таблица рекордов - показывает 10 лучших результатов игры. Если игрок не вошел в 10 лучших его результат показывает вместо 10 результата, с указанием его места в таблице рекордов.
* Режима теста - режим в котором таймер обратного отсчета не запускается, и игра не останавливается при взаимодействиях, которые подразумевают проигрыш или конец игры
* Реализация логики работы игрового поля, например, изменение количества собранных объектов, уменьшение количества “жизней”, уменьшение оставшегося времени и т.д., а также обработка соответствующих событий - конец игры, проигрыш, выигрыш при наступлении определенных действий.

1. В данном задании участнику необходимо разработать игру «Пенальти», для футбольных болельщиков. Цель игры – забить гол в ворота соперника. Игрок выбирает область ворот, в которую будет осуществлен удар, после чего появляется показатель сила удара, меняющийся от максимального к минимальному каждые 300 мс. Во время повторного клика сила удара фиксируется и мяч перемещается в указанную область. В створе ворот должен размещаться вратарь команды противника, который перемещается от левой границы ворот к правой каждые 1000 мс. Если при ударе по воротам мяч попадает во вратаря или в зону вокруг него, равную половине ширины вратаря, вратарь ловит мяч и пенальти считается проигранным. В обратном случае, гол и победа засчитывается за игроком. Время на пенальти ограничено таймером. В случае трех голов подряд в игре должен появляться бонус – ускорение мяча при ударе в два раза. С каждым голом скорость перемещения вратаря уменьшается на 10 мс.

**Стартовый экран** - содержит наименование игры, поле ввода своего имени, а также кнопку “Начать игру”. Имя не может быть пустым. Кнопка “Начать игру” активна, если только введено имя пользователя.

**Экран игры** - интерфейс должен содержать:

* блок с таймером - содержит таймер обратного отсчета с указанием минут, секунд в формате мм:сс. Таймер должен обновляться без перезагрузки страницы. В этом же блоке располагается кнопка “пауза”
* блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
* блок с именем игрока - если игрок на стартовом экране в имени ввел “tester” то имя пользователя должно отображаться по другому, так как это “кодовое” слово для игры в тестовом режиме.
* игровое поле - в зависимости от выбранной экспертами игры. Процесс игры должен включать в себя взаимодействия пользователя и компьютера и протекать без обновления страницы

**Экран окончания игры** - содержит результат игрока.

Реализуйте логику, в состав которой должны быть включены следующие функции:

* Пауза игрового процесса - останавливается время на таймере, запрещается воздействие на игровое поле, перечень элементов, для которых может сохранять анимация описывается экспертами отдельно (например, перемещение фона в режиме паузы). Режим паузы также может быть инициирован по нажатию на клавишу пробел. Возобновление игры так же возможно по нажатию на клавишу “пробел” или по нажатию на кнопку “Пауза” на игровом поле
* Таймер обратного отсчета - начинает обратный отсчет с началом игры, как только доходит до значения 00:00 игра заканчивается
* Режима теста - режим в котором таймер обратного отсчета не запускается, и игра не останавливается при взаимодействиях, которые подразумевают проигрыш или конец игры
* Реализация логики работы игрового поля, например, изменение количества собранных объектов, уменьшение количества “жизней”, уменьшение оставшегося времени и т.д., а также обработка соответствующих событий - конец игры, проигрыш, выигрыш при наступлении определенных действий.

1. Вам необходимо разработать онлайн игру «SpaceShip» (космический симулятор).

Цель игры – сбить как можно больше пролетающих мимо метеоритов, не допустив при этом столкновения с кораблем. Поле имеет прямоугольную форму, корабль может двигаться влево и вправо. Метеориты могут двигаться под произвольным углом к нижней границе игрового поля. Корабль имеет функцию выстрела, которая уничтожает первый метеорит на текущей вертикали. Количество выстрелов неограниченно. Если метеорит попадает в корабль – игра заканчивается. За сбитый выстрелом метеорит начисляются очки, за «пропущенный» метеорит (объект дошел до нижней горизонтали) очки вычитаются.

**Стартовый экран** - содержит наименование игры, поле ввода своего имени, а также кнопку “Начать игру”. Имя не может быть пустым. Кнопка “Начать игру” активна, если только введено имя пользователя.

**Экран игры** - интерфейс должен содержать:

* блок с таймером - содержит таймер обратного отсчета с указанием минут, секунд в формате мм:сс. Таймер должен обновляться без перезагрузки страницы. В этом же блоке располагается кнопка “пауза”
* блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
* блок с именем игрока - если игрок на стартовом экране в имени ввел “tester” то имя пользователя должно отображаться по другому, так как это “кодовое” слово для игры в тестовом режиме.
* игровое поле - в зависимости от выбранной экспертами игры. Процесс игры должен включать в себя взаимодействия пользователя и компьютера и протекать без обновления страницы

**Экран окончания игры** - содержит результат игрока.

Реализуйте логику, в состав которой должны быть включены следующие функции:

* Пауза игрового процесса - останавливается время на таймере, запрещается воздействие на игровое поле, перечень элементов, для которых может сохранять анимацию, описывается экспертами отдельно (например, перемещение фона в режиме паузы). Режим паузы также может быть инициирован по нажатию на клавишу пробел. Возобновление игры так же возможно по нажатию на клавишу “пробел” или по нажатию на кнопку “Пауза” на игровом поле
* Таймер обратного отсчета - начинает обратный отсчет с началом игры, как только доходит до значения 00:00 игра заканчивается
* Реализация логики работы игрового поля, например, изменение количества собранных объектов, уменьшение количества “жизней”, уменьшение оставшегося времени и т.д., а также обработка соответствующих событий - конец игры, проигрыш, выигрыш при наступлении определенных действий.

1. Вам необходимо разработать игру по типу морского боя. Игра представляет собой два поля, разделенных на квадратные области, в которых находятся фрагменты «кораблей». В одном из полей находятся «корабли» игрока, а в другом – скрытые «корабли» противника(компьютера). После размещения «кораблей» в полях, их вращение или изменение местоположения – запрещено. Задача игрока – «потопить» все «корабли» противника до полной потери собственных. Игрок может сделать только один ход подряд, следующий ход возможен после хода компьютера.

**Стартовый экран** - содержит наименование игры, поле ввода своего имени, а также кнопку “Начать игру”. Имя не может быть пустым. Кнопка “Начать игру” активна, если только введено имя пользователя.

**Экран игры** - интерфейс должен содержать:

* блок с таймером - содержит таймер продолжительности игры с указанием минут, секунд в формате мм:сс. Таймер должен обновляться без перезагрузки страницы. В этом же блоке располагается кнопка “пауза”
* блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
* Процесс игры должен включать в себя взаимодействия пользователя и компьютера и протекать без обновления страницы

**Экран окончания игры** - содержит сообщение о том кто и с каким счетом выиграл.

Реализуйте логику, в состав которой должны быть включены следующие функции:

* Пауза игрового процесса - останавливается время на таймере, запрещается воздействие на игровое поле, перечень элементов, для которых может сохранять анимация описывается экспертами отдельно (например, перемещение фона в режиме паузы). Режим паузы также может быть инициирован по нажатию на клавишу пробел. Возобновление игры так же возможно по нажатию на клавишу “пробел” или по нажатию на кнопку “Пауза” на игровом поле
* Таймер продолжительности игрового процесса - начинает счет с началом игры, останавливается, когда заканчивается игра.
* Реализация логики работы игрового поля, например, расстановка «кораблей», способ выбора целей, и т.д.

1. Вам необходимо разработать онлайн игру (аркаду) – «Гонки». Участнику необходимо реализовать ограниченное по размерам игровое поле, на котором в нижней части передвигается машина. В верхней части поля располагаются объекты, которые являются препятствиями или бонусами в игре. Скорость перемещения объектов увеличивается со временем игры. Задача игрока – продержаться в игре как можно больше времени.

**Стартовый экран** - содержит наименование игры, поле ввода своего имени, а также кнопку “Начать игру”. Имя не может быть пустым. Кнопка “Начать игру” активна, если только введено имя пользователя.

**Экран игры** - интерфейс должен содержать:

* блок с таймером - содержит таймер обратного отсчета с указанием минут, секунд в формате мм:сс. Таймер должен обновляться без перезагрузки страницы. В этом же блоке располагается кнопка “пауза”
* блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
* блок с именем игрока - если игрок на стартовом экране в имени ввел “terter” то имя пользователя должно отображаться по другому, так как это “кодовое” слово для игры в тестовом режиме.
* игровое поле - в зависимости от выбранной экспертами игры. Процесс игры должен включать в себя взаимодействия пользователя и компьютера и протекать без обновления страницы

**Экран окончания игры** - содержит результат игрока.

Реализуйте логику, в состав которой должны быть включены следующие функции:

* Пауза игрового процесса - останавливается время на таймере, запрещается воздействие на игровое поле, перечень элементов, для которых может сохранять анимацию, описывается экспертами отдельно (например, перемещение фона в режиме паузы). Режим паузы также может быть инициирован по нажатию на клавишу пробел. Возобновление игры так же возможно по нажатию на клавишу “пробел” или по нажатию на кнопку “Пауза” на игровом поле
* Таймер обратного отсчета - начинает обратный отсчет с началом игры, как только доходит до значения 00:00 игра заканчивается
* Режима теста - режим в котором таймер обратного отсчета не запускается, и игра не останавливается при взаимодействиях, которые подразумевают проигрыш или конец игры
* Реализация логики работы игрового поля, например, изменение количества собранных объектов, уменьшение количества “жизней”, уменьшение оставшегося времени и т.д., а также обработка соответствующих событий - конец игры, проигрыш, выигрыш при наступлении определенных действий.

1. Вам необходимо разработать браузерную игру.

**Описание игры**

Игровое поле представляет собой сад с падающими фруктами. В определённой области через заданный промежуток времени появляется какой-либо фрукт. Игрок должен набрать максимальное количество баллов, кликая на такие же фрукты. После клика по нужному фрукту, он исчезает, а игрок получает очки. Если игрок кликнул не на тот фрукт, то фрукт исчезает, а игрок теряет очки. Каждыйфрукт движется со случайной скоростью сверху вниз. Если по фрукту не кликнули, и он достиг нижней части игрового поля, то он исчезает. Одновременно на одном игровом поле может находиться не более 10 фруктов.

**Стартовый экран** - содержит наименование игры, поле ввода имени игрока, а также кнопку “Начать игру”. Имя не может быть пустым. Кнопка “Начать игру” активна, если только введено имя пользователя.

На стартовом экране представлена инструкция к игре.

**Экран игры** - интерфейс должен содержать:

* блок с таймером - содержит таймер обратного отсчета с указанием минут, секунд в формате мм:сс. Таймер должен обновляться без перезагрузки страницы. В этом же блоке располагается кнопка “пауза”
* блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
* блок с именем игрока.
* блок, в котором отображается фрукт, по которым нужно кликать в данный момент времени
* игровое поле - Процесс игры должен включать в себя взаимодействия пользователя и компьютера и протекать без обновления страницы

**Экран окончания игры** - содержит набранное количество очков.

Реализация логики, в состав которой должны быть включены следующие функции:

* Пауза игрового процесса - останавливается время на таймере, запрещается воздействие на игровое поле. Возобновление игры также возможно по нажатию на кнопку “Пауза” на игровом поле.
* Таймер обратного отсчета - начинает обратный отсчет с началом игры, начальное значение 00:30. Как только таймер обратного отсчета доходит до значения 00:00, игра заканчивается.

### Критерии оценивания

3: Изучен теоретический материал 5-6 главы. Выполнено задание (возможно реализация не всех функций). Составлен отчет о работе;

4: Изучен теоретический материал 5-6 главы. Выполнено задание (возможно с незначительными погрешностями). Составлен отчет о работе;

5: Изучен теоретический материал 5-6 главы. Выполнено задания. Составлен отчет о работе. Присутствует единый стиль оформления кода, комментарии, улучшен интерфейс программы.

### 

# 7. Использованные источники

1. Современный учебник JavaScript [Электронный ресурс] – Режим доступа: https://learn.javascript.ru Дата обращения: 1.01.2020