### 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*

### Проверочная 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.

### 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:

1. Хранить «текущее значение» в свойстве value. Начальное значение устанавливается в аргументе конструктора startingValue.
2. 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:

1. push добавляет элемент в конец.
2. 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.

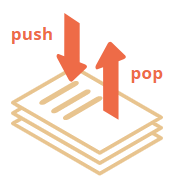
Существует и другой вариант применения для массивов – структура данных, называемая *стек*.

Она поддерживает два вида операций:

1. push добавляет элемент в конец.
2. 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:

1. arr.indexOf(item, from) ищет item, начиная с индекса from, и возвращает индекс, на котором был найден искомый элемент, в противном случае -1.
2. arr.lastIndexOf(item, from) – то же самое, но ищет справа налево.
3. 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:

1. item – очередной элемент.
2. index – его индекс.
3. 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:
2. ввод заранее заданного количества объектов;
3. ввод до появления объекта с заданным признаком;
4. диалог с пользователем о необходимости продолжать ввод.
5. Функцию просмотра содержимого массива объектов.
6. Функцию дополнения уже существующего массива новыми объектами.
7. Функцию поиска и вывода на экран объекта (объектов) с заданным значением свойства.
8. 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

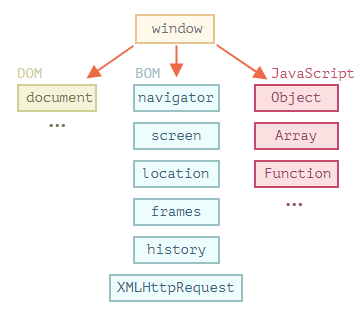
## Браузерное окружение

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:

1. 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).
2. 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**>

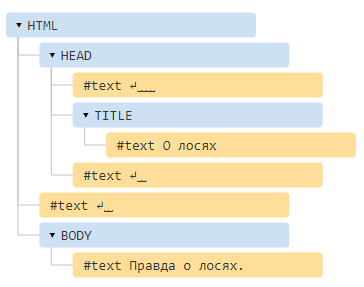
<**body**>

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:

1. перевод строки: ↵ (в JavaScript он обозначается как \n)
2. 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**>

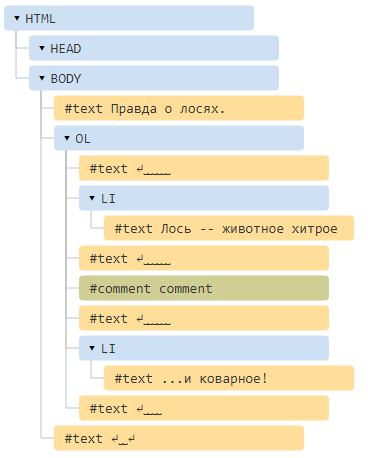
*<!-- комментарий -->*

<**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.

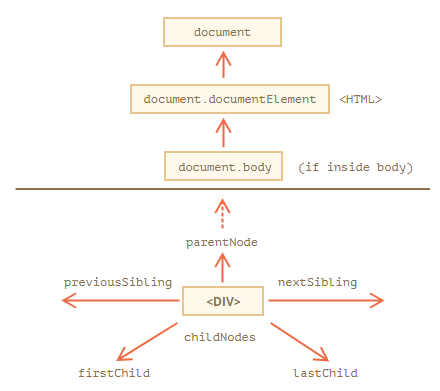
**Все, что есть в HTML, даже комментарии, является частью DOM.**

## 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:

1. **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>.
2. **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:

1. Для перебора коллекции мы можем использовать for..of;
2. 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**>

1. говорят, что <body> – «следующий» или «правый» сосед <head>
2. 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>*

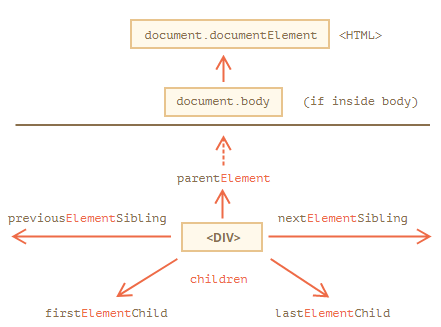
*alert*( ***document***.**body**.**previousSibling** ); *// HTMLHeadElement*

### 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:

1. children – коллекция детей, которые являются элементами.
2. firstElementChild, lastElementChild – первый и последний дочерний элемент.
3. previousElementSibling, nextElementSibling – соседи-элементы.
4. parentElement – родитель-элемент.

### Практическая работа №14

1. Для страницы:

<**html**>

<**body**>

<**div**>Пользователи:</**div**>

<**ul**>

<**li**>Джон</**li**>

<**li**>Пит</**li**>

</**ul**>

</**body**>

</**html**>

1. Напишите код, который получит элемент <div>
2. Напишите код, который получит <ul>
3. 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.

1. elem.getElementsByTagName(tag) ищет элементы с данным тегом и возвращает их коллекцию. Передав "\*" вместо тега, можно получить всех потомков.
2. elem.getElementsByClassName(className) возвращает элементы, которые имеют данный CSS-класс.
3. 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.

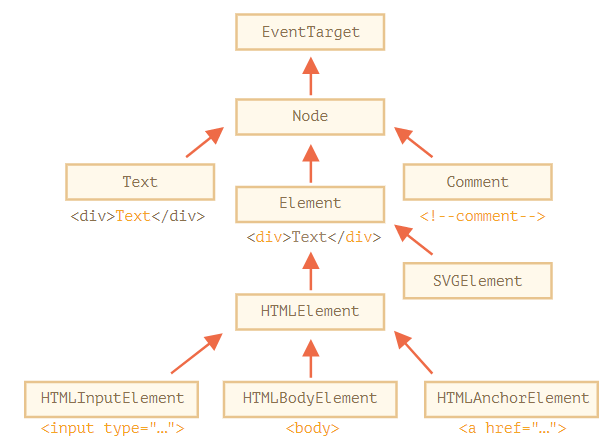
### Классы DOM-узлов

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:

1. *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.
2. *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.
3. *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.
4. *HTMLElement* – является базовым классом для всех остальных HTML-элементов. От него наследуют конкретные элементы:
   1. HTMLInputElement – класс для тега <input>,
   2. HTMLBodyElement – класс для тега <body>,
   3. HTMLAnchorElement – класс для тега <a>,
   4. …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.

1. Свойство tagName есть только у элементов Element.
2. Свойство nodeName определено для любых узлов Node:
   1. для элементов оно равно tagName.
   2. 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:

1. value – значение для <input>, <select> и <textarea> (HTMLInputElement, HTMLSelectElement…).
2. href – адрес ссылки «href» для <a href="..."> (HTMLAnchorElement).
3. id – значение атрибута «id» для всех элементов (HTMLElement).
4. …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 №15

1. У нас есть дерево, структурированное как вложенные списки ul/li.

Напишите код, который выведет каждый элемент списка <li>:

1. Какой в нём текст (без поддерева) ?
2. 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:

1. Им можно присвоить любое значение.
2. 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:

1. elem.hasAttribute(name) – проверяет наличие атрибута.
2. elem.getAttribute(name) – получает значение атрибута.
3. elem.setAttribute(name, value) – устанавливает значение атрибута.
4. elem.removeAttribute(name) – удаляет атрибут.

У HTML-атрибутов есть следующие особенности:

1. Их имена регистронезависимы (id то же самое, что и ID).
2. 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 №16

1. Сделайте все внешние ссылки оранжевыми, изменяя их свойство style.

Ссылка является внешней, если:

1. Её href содержит ://
2. 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**>

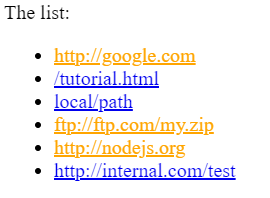
*// добавление стиля для одной ссылки*

**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:

1. node.append(...nodes or strings) – добавляет узлы или строки в конец node,
2. node.prepend(...nodes or strings) – вставляет узлы или строки в начало node,
3. node.before(...nodes or strings) –- вставляет узлы или строки до node,
4. node.after(...nodes or strings) –- вставляет узлы или строки после node,
5. 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.

1. 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.

### Проверочная работа №17

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:

1. Запрашивайте содержимое пункта у пользователя с помощью prompt.
2. Создавайте элемент <li> и добавляйте его к <ul>.
3. 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:

1. elem.classList.add/remove("class") – добавить/удалить класс.
2. elem.classList.toggle("class") – добавить класс, если его нет, иначе удалить.
3. 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'**);

### Проверочная работа №18

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**;

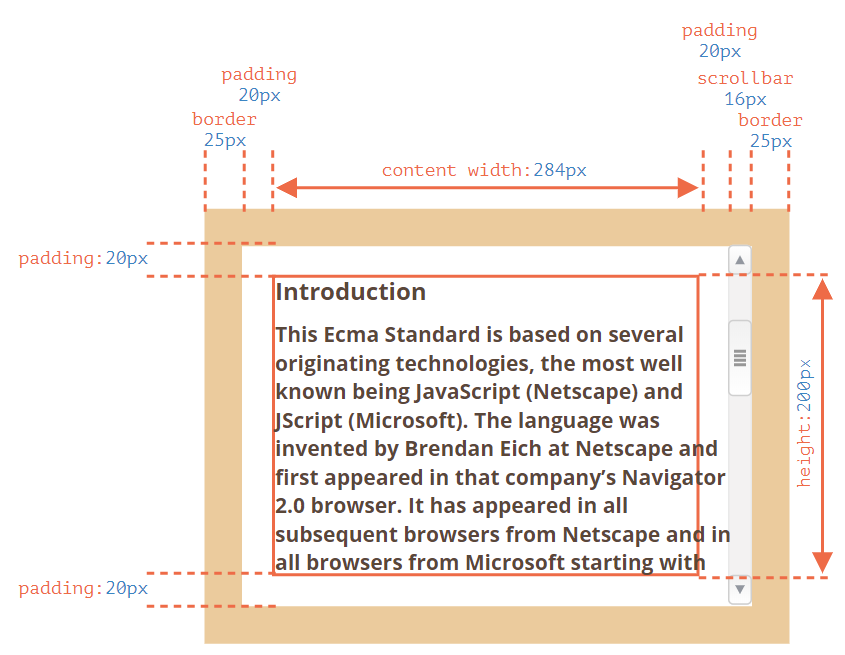
**overflow**: **auto**;

}

</**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

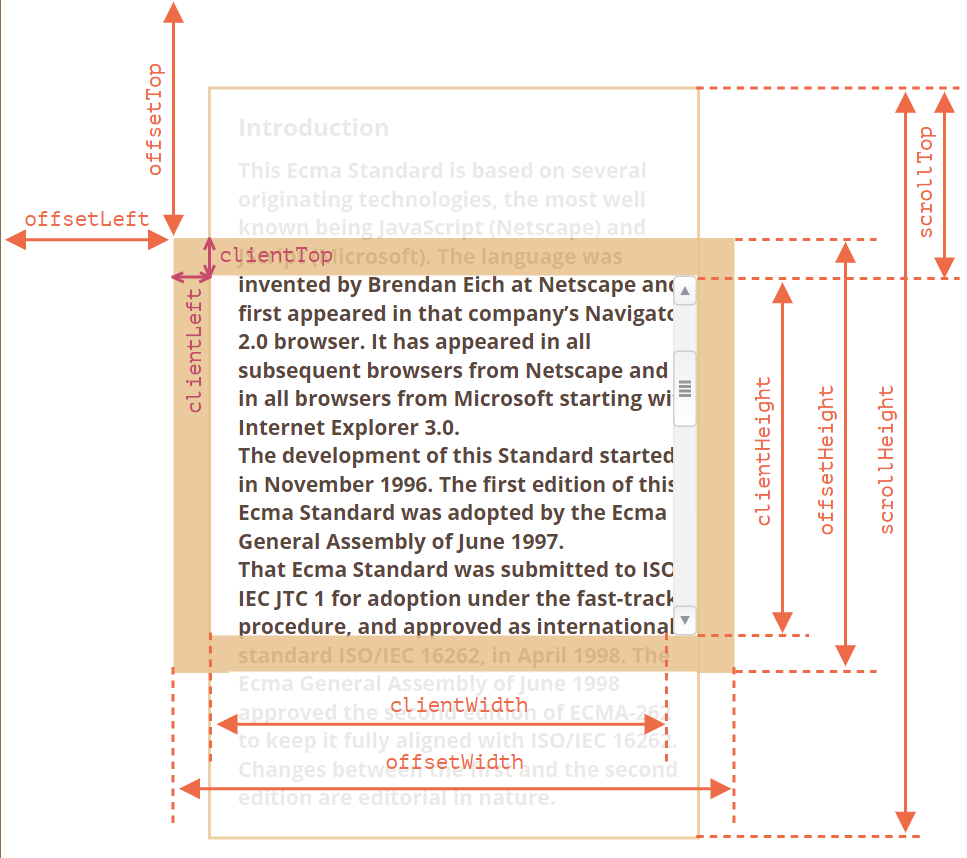
Эти свойства редко используются, но так как они являются «самыми внешними» метриками, мы начнём с них.

В свойстве offsetParent находится предок элемента, который используется внутри браузера для вычисления координат при рендеринге.

То есть, ближайший предок, который удовлетворяет следующим условиям:

1. Является CSS-позиционированным (CSS-свойство position равно absolute, relative, fixed или sticky),
2. или <td>, <th>, <table>,
3. или <body>.

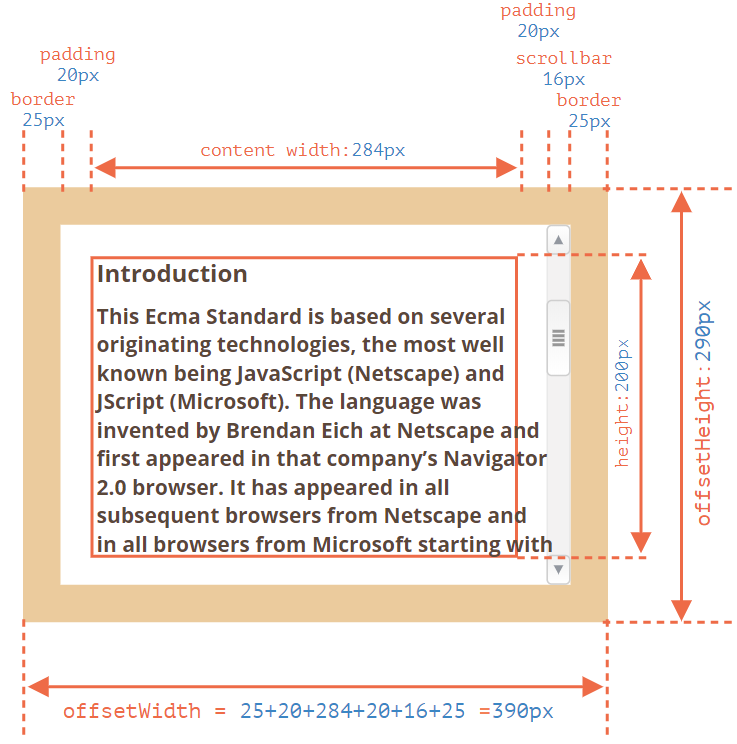
Свойства offsetLeft/offsetTop содержат координаты x/y относительно верхнего левого угла offsetParent.



**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).

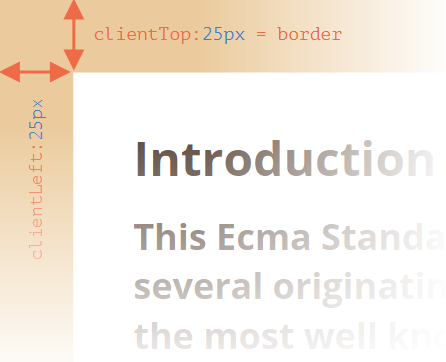


**Рисунок 5.9.** offsetWidth/Height

### clientTop/Left

Пойдём дальше. Внутри элемента у нас рамки (border).

Для них есть свойства-метрики clientTop и clientLeft (Рисунок 5.10).

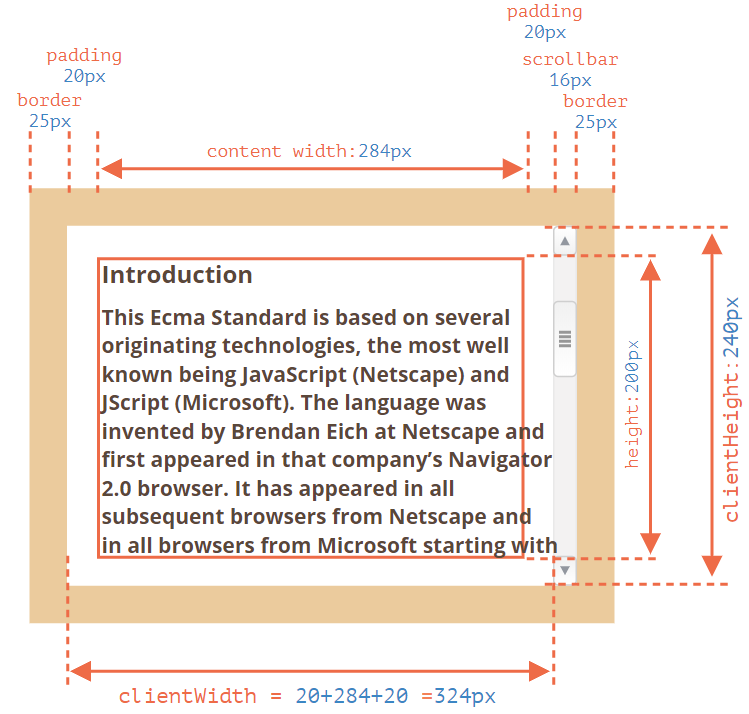


**Рисунок 5.10.** clientTop/Left

### 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

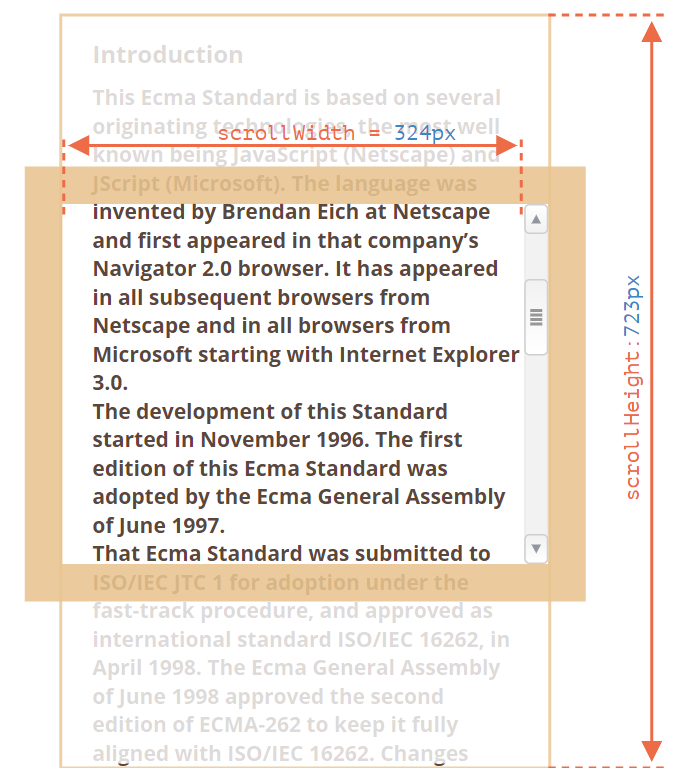
Эти свойства – как clientWidth/clientHeight, но также включают в себя прокрученную (которую не видно) часть элемента (Рисунок 5.12).

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.
   1. 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.
   1. 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:

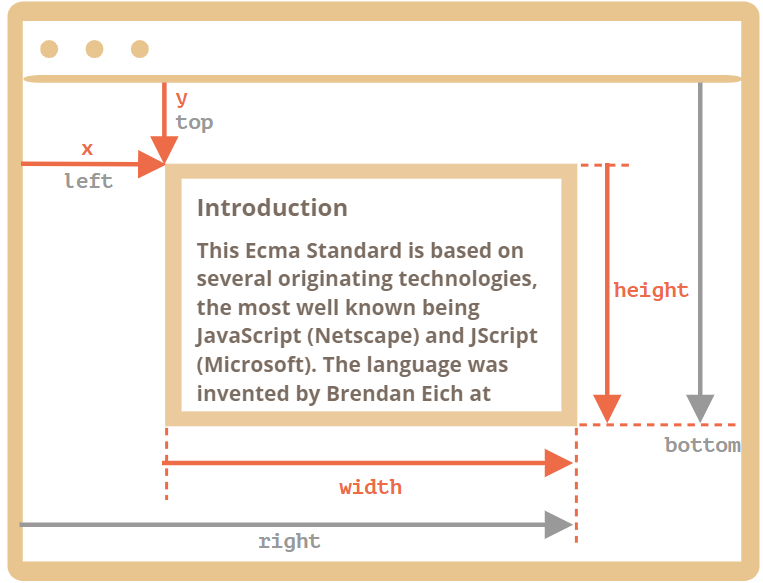
1. x/y – X/Y-координаты начала прямоугольника относительно окна,
2. width/height – ширина/высота прямоугольника (могут быть отрицательными).

Дополнительные, «зависимые», свойства:

1. top/bottom – Y-координата верхней/нижней границы прямоугольника,
2. 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:

1. pageY = clientY + высота вертикально прокрученной части документа.
2. 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 №19

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:**

1. click – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании).
2. contextmenu – происходит, когда кликнули на элемент правой кнопкой мыши.
3. mouseover / mouseout – когда мышь наводится на / покидает элемент.
4. mousedown / mouseup – когда нажали / отжали кнопку мыши на элементе.
5. mousemove – при движении мыши.

**События на элементах управления:**

1. submit – пользователь отправил форму <form>.
2. focus – пользователь фокусируется на элементе, например нажимает на <input>.

**Клавиатурные события:**

1. keydown и keyup – когда пользователь нажимает / отпускает клавишу.

**События документа**:

1. DOMContentLoaded – когда HTML загружен и обработан, DOM документа полностью построен и доступен.

**CSS events:**

1. 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:

1. Клик по ссылке инициирует переход на новый URL.
2. Нажатие на кнопку «отправить» в форме – отсылку её на сервер.
3. 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:

1. Основной способ – это воспользоваться объектом event. Для отмены действия браузера существует стандартный метод event.preventDefault().
2. 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:

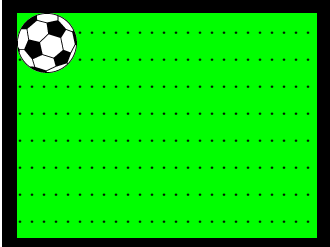
<**a href="/" onclick="return false"**>Нажми здесь</**a**>

или

<**a href="/" onclick="*event***.preventDefault()**"**>здесь</**a**>

### Практическая работа №20

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:

1. Центр мяча должен совпадать с местом нажатия мыши (если это возможно без пересечения краёв поля);
2. CSS-анимация желательна, но не обязательна;
3. Мяч ни в коем случае не должен пересекать границы поля;
4. При прокрутке страницы ничего не должно ломаться;

Заметки:

1. Код должен уметь работать с различными размерами мяча и поля, не привязываться к каким-либо фиксированным значениям.
2. Используйте свойства event.clientX/event.clientY для определения координат мыши при клике.
3. 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:

1. event.which == 1 – левая кнопка
2. event.which == 2 – средняя кнопка
3. 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:

1. shiftKey: Shift
2. altKey: Alt (или Opt для Mac)
3. ctrlKey: Ctrl
4. 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.

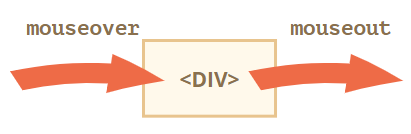
### Практическая работа №21

1. Создайте список, в котором элементы могут быть выделены, как в файловых менеджерах.
2. При клике на элемент списка выделяется только этот элемент (добавляется класс .selected), отменяется выделение остальных элементов.
3. 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:

1. event.target – это элемент, на который курсор перешёл.
2. event.relatedTarget – это элемент, с которого курсор ушёл (relatedTarget → target).

Для события mouseout наоборот:

1. event.target – это элемент, с которого курсор ушёл.
2. 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.

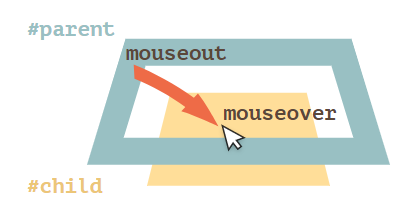
Браузер периодически проверяет позицию курсора и, заметив изменения, генерирует события mousemove.

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:

1. куда пользователь положил элемент в конце переноса, чтобы обработать его окончание
2. 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 №22

1. Создайте слайдер:



Важные детали:

1. Слайдер должен нормально работать при резком движении мыши влево или вправо за пределы полосы. При этом бегунок должен останавливаться чётко в нужном конце полосы.
2. 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:

1. Появление символа (самое очевидное).
2. Удаление символа (клавиша Delete).
3. Прокрутка страницы (клавиша PageDown).
4. Открытие диалогового окна браузера «Сохранить» (Ctrl+S)
5. …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:

1. стартовый экран - содержит наименование игры, поле ввода своего имени, а также кнопку “Начать игру”. Кнопка “Начать игру” активна, если введено имя пользователя.
2. экран игры - интерфейс должен содержать:
   1. блок с таймером - содержит таймер обратного отсчета с указанием минут, секунд в формате мм:сс.
   2. блок с заработанными баллами (количество оставшихся секунд по таймеру) - количество заработанных игроком баллов за текущую игру.
   3. блок с руководством по игре.
3. экран окончания игры - содержит результат игрока и кнопку, позволяющую начать заново.
4. 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:

1. 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”
2. блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
3. 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.
4. 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:

1. 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
2. Таймер обратного отсчета - начинает обратный отсчет с началом игры, как только доходит до значения 00:00 игра заканчивается
3. Режима теста - режим в котором таймер обратного отсчета не запускается, и игра не останавливается при взаимодействиях, которые подразумевают проигрыш или конец игры
4. 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.
5. 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:**

1. ввод имени игроков перед началом игры
2. по клику на рыбку она исчезает, а игроку начисляются очки
3. рыбки двигаются
4. игра заканчивается с окончанием времени

**Дополнительный функционал программы:**

1. присутствуют несколько видов рыб разных размеров и видов, при клике на каждую начисляется разное количество очков
2. присутствуют анимации появления и исчезновения рыб, анимации таймера, элементов интерфейса
3. после окончания игры появляется таблица рекордов, в которой показан текущий результат игрока и результаты лучших 10 попыток.
4. 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:**

1. присутствует разработанный дизайн и адаптивная верстка
2. можно создавать блокноты и списки дел в них
3. можно отмечать выполненные дела, удалять и редактировать заметки и блокноты
4. все данные сохраняются после обновления страницы

**Дополнительный функционал программы:**

1. можно окрашивать блокноты в разные цвета
2. можно осуществлять поиск по заметкам
3. устанавливать приоритет дел в списке
4. сортировать дела по приоритету, названию, дате
5. присутствует анимация интерфейса
6. 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:

1. стартовый экран - содержит наименование игры, поле ввода своего имени, а также кнопку “Начать игру”. Кнопка “Начать игру” активна, если введено имя пользователя.
2. экран игры - интерфейс должен содержать:
   1. блок с таймером - содержит таймер обратного отсчета с указанием минут, секунд в формате мм:сс.
   2. блок с заработанными баллами (количество оставшихся секунд по таймеру) - количество заработанных игроком баллов за текущую игру.
   3. блок с руководством по игре.
3. экран окончания игры - содержит результат игрока и кнопку, позволяющую начать заново.
4. 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:

1. стартовый экран - содержит наименование игры, поле ввода своего имени, а также кнопку “Начать игру”. Кнопка “Начать игру” активна, если введено имя пользователя.
2. экран игры - интерфейс должен содержать:
   1. блок с таймером - содержит таймер обратного отсчета с указанием минут, секунд в формате мм:сс.
   2. блок с заработанными баллами (количество оставшихся секунд по таймеру) - количество заработанных игроком баллов за текущую игру.
   3. блок с руководством по игре.
3. экран окончания игры - содержит результат игрока и кнопку, позволяющую начать заново.
4. 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:

1. 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”
2. блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
3. 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.
4. 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:

1. 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
2. Таймер обратного отсчета - начинает обратный отсчет с началом игры, как только доходит до значения 00:00 игра заканчивается
3. Режима теста - режим в котором таймер обратного отсчета не запускается, и игра не останавливается при взаимодействиях, которые подразумевают проигрыш или конец игры
4. 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.
5. 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:

1. 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»
2. блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
3. 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.
4. 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:

1. 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
2. Таймер обратного отсчета - начинает обратный отсчет с началом игры, как только доходит до значения 00:00 игра заканчивается
3. 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.
4. Режима теста - режим в котором таймер обратного отсчета не запускается, и игра не останавливается при взаимодействиях, которые подразумевают проигрыш или конец игры
5. 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.
6. 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:

1. 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”
2. блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
3. 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.
4. 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:

1. 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
2. Таймер обратного отсчета - начинает обратный отсчет с началом игры, как только доходит до значения 00:00 игра заканчивается
3. 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.
4. Режима теста - режим в котором таймер обратного отсчета не запускается, и игра не останавливается при взаимодействиях, которые подразумевают проигрыш или конец игры
5. 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.
6. 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:

1. 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”
2. блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
3. 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.
4. 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:

1. 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
2. Таймер обратного отсчета - начинает обратный отсчет с началом игры, как только доходит до значения 00:00 игра заканчивается
3. Режима теста - режим в котором таймер обратного отсчета не запускается, и игра не останавливается при взаимодействиях, которые подразумевают проигрыш или конец игры
4. 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.
5. 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:

1. 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”
2. блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
3. 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.
4. 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:

1. 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
2. Таймер обратного отсчета - начинает обратный отсчет с началом игры, как только доходит до значения 00:00 игра заканчивается
3. 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.
4. 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:

1. 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”
2. блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
3. 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:

1. 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
2. Таймер продолжительности игрового процесса - начинает счет с началом игры, останавливается, когда заканчивается игра.
3. Реализация логики работы игрового поля, например, расстановка «кораблей», способ выбора целей, и т.д.
4. 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:

1. 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”
2. блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
3. 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.
4. 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:

1. 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
2. Таймер обратного отсчета - начинает обратный отсчет с началом игры, как только доходит до значения 00:00 игра заканчивается
3. Режима теста - режим в котором таймер обратного отсчета не запускается, и игра не останавливается при взаимодействиях, которые подразумевают проигрыш или конец игры
4. 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.
5. 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:

1. 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”
2. блок с заработанными баллами - количество заработанных игроком баллов за текущую игру. Обнуляется при начале новой игры.
3. блок с именем игрока.
4. блок, в котором отображается фрукт, по которым нужно кликать в данный момент времени
5. 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:

1. Пауза игрового процесса - останавливается время на таймере, запрещается воздействие на игровое поле. Возобновление игры также возможно по нажатию на кнопку “Пауза” на игровом поле.
2. 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