JAVASKRIPT

|  |  |
| --- | --- |
| **prompt()** | Даёт *JS* возможность выводить специальное окно (*pop-up*), куда пользователь вводит какое-то значение. prompt только присваивает значение, но никак его не обрабатывает. |
| **alert()** | Используется для вывода результата обработки или любой другой информации на экран пользователя при помощи специального окна. Если нужно вывести информацию, содержащую пояснение или любой другой текст, то её надо брать в одинарные кавычки (' '). Результат обработки, то есть переменную, в кавычки брать не следует. |

Boolean — это тип данных, переменные которого принимают одно из значений: true (**истина**, «да», логическая единица «1»); false (ложь, «нет», логический ноль «0»).

Как вы знаете, **число** **в** **кавычках** представляет собой строку. Например, '3' - это строка. Однако, при сравнении таких строк с настоящими **числами** **JavaScript** считает, что строка **в** **кавычках** равна такому же **числу**. Давайте для примера сравним строку '3' и **число** 3: if ('3' == 3) {alert('верно'); // сработает этот alert - **значения** равны} else {alert('неверно');}

Итак*, JavaScript* определяет 8 встроенных типов данных:

* + null
  + undefined
  + boolean
  + number
  + string
  + symbol
  + object
  + bigInt

Все эти данные, кроме object, являются «простыми» или «примитивными» (их ещё называют [скалярные](https://plsql.ru/plsql/scalar/)).

**Примеры валидных значений:**

|  |  |
| --- | --- |
| **42** | number |
| **'Hello world'** | string |
| **false** | boolean |
| **undefined** | undefined |
| **null** | null |
| **[1,2,3,4]** | array (object) |
| **{a:2, b:3}** | object |
| **1234564629123456789087493828901234567890n** | bigInt |

Преобразование к логическому типу происходит в логическом контексте или вызывается логическими операторами (&&, ||, !). Такое преобразование называется **неявным**.

|  |  |
| --- | --- |
| **Возвращают***false* | **Возвращают***true* |
| 0, -0, NaN, пустая строка в любых кавычках, null, undefined, false | Все остальные значения, включая отрицательные числа, пустой массив или объект. |

Операторы могут быть разбиты на категории по **количеству** требуемых им операндов:

|  |  |
| --- | --- |
| **унарные** | *operator* operand;  ++a;  operand *operator*;  a++; |
| **бинарные** | operand *operator* operand;  a + b; a \* b; |
| **тернарные** | на данный момент существует только один — условный оператор;  operand operator operand operator operand;  a > b ? a : b |

**В JS используются три условных оператора:**

* 1. if-else
  2. switch
  3. тернарный

1. **if-else**
2. **if** (condition)
3. statement1
4. [**else**
5. statement2]
6. *сondition* —  выражение, результат которого либо *true*, либо *false*. Можно записывать выражения, либо значения. Происходит неявное преобразование к логическому типу.
7. *statement1* — инструкции, которые будут выполнены, если *condition* возвращает *true*.
8. *statement2* — инструкции, которые будут выполнены, если *condition* возвращает *false* и присутствует слово *else*. Наличие этой части не обязательно.

**Важно!** Когда в блоках if или else всего одно действие, скобки можно не ставить. При двух и более — скобки обязательны. Лучше привыкнуть и записывать со скобками, так вы обезопасите себя от будущих проблем.

Оператор switch используется, чтобы проверить несколько возможных значений одного выражения.

У этой конструкции есть несколько блоков case и необязательный блок default.

В каждом case, кроме последнего, нужен оператор break, иначе после выполнения всех утверждений в каком-то конкретно case исполнение перейдёт к следующему блоку case без проверки условия.

Важно помнить, что в case необходимо размещать значение выражения, а не логическую операцию. Пример ниже не будет работать корректно:

**switch** (result) {

**case** (result < **0**):

console.log('результат меньше 0');

**break**;

**default**:

console.log('результат больше или равен 0');

}

**Тернарный оператор**

Уникальный в своем роде оператор, так как состоит из трёх операндов (отсюда и название).

let result = условие ? значение1 : значение2;

«Тернарка» состоит из условия (= if(условие)), затем символа ? (= отправляем в блок инструкции, если условие истинно) и затем : (= отправляем в блок инструкции, если условие ложно).

**let** text = ''

// 1 вариант

**if** (result < **4**) {

text ="результат меньше 4"

} **else** {

text ="результат больше или равен 4"

}

// 2 вариант

text = result < **4** ? "результат меньше 4" : "результат больше или равен 4"

Видим, как уменьшилась длина кода. Но есть важное отличие — тернарный оператор**всегда** возвращает значение, поэтому мы должны записать результат в переменную. То есть, мы не можем записать вот эту конструкцию через этот оператор:

**if** (result < **4**) {

console.log("результат меньше 4")

} **else** {

console.log("результат больше или равен 4")

}

Если нужно сравнивать несколько значений одного выражения, например, чему равна переменная **result**, то подойдёт **switch**.

2

Если таких выражений меньше четырех, лучше использовать **if-else**, так как он работает быстрее **switch** на маленьком числе проверок.

3

Если нужно выполнять какие-то операции для определения истинности условия, например, сравнивать одно выражение с другим — **result === 2**, подойдёт **if-else**.

4

Если нужно записать некоторое значение в переменную после проверки одного условия, подойдёт тернарный.

● Имя переменной должно содержать только буквы, цифры или символы $ и \_. ● Первый символ не должен быть цифрой. ● Имя переменной не должно совпадать с зарезервированным ключевым словом (class, var, for, is, function и т.д.) Рекомендации по выбору имени для переменной: Имя должно быть осмысленным, т.е. отображать суть значения, которое содержится в переменной Имена переменных записываются в нотации camel case: isUserLogged (кроме констант)

module5\_homework\_Nalivaiko\_Sergey

<https://github.com/Egnat/module5_homework-Nalivaiko-Sergey>

https://github.com/Egnat/module5\_homework\_Nalivaiko\_Sergey

**Задание 1.**

Напишите программу, которая работала бы следующим образом: в prompt вводится значение. С помощью унарного плюса (арифметический оператор) необходимо преобразовать его в число, затем проверить с помощью typeof, принадлежит ли оно к множеству Number.

Если это число, то вывести в консоль чётное оно или нечётное.

Если передано не число, выведите: «Упс, кажется, вы ошиблись».

\*NaN, хоть и относится к типу Number, числом не является. Добавьте отдельную проверку для этого значения.

Ответы

/\*let meaning = +prompt("Введите значение");

if (isNaN(meaning)) {

console.log("Вы ошиблись");

} else if(meaning %2 !== 0){

console.log("не четное");

} else if (meaning %2 == 0) {

console.log("четное");

}

console.log(typeof meaning)\*/

let meaning = +prompt("Введите число");

if(typeof meaning === "number" && !Number.isNaN(meaning)) {

if ( meaning %2 === 0) {

console.log("это четное число");

} else {

console.log("Это нечетное число"); }

}

else {

console.log ("Упс, кажется, Вы ошиблись");

}

**Задание 2.**

Дана переменная Х, которая может принимать любое значение. Написать программу, которая в зависимости от типа данных Х выводит в консоль сообщение вида: «X — число».

Опишите три случая: когда х = числу, строке или логическому типу. В других случаях выводите сообщение: «Тип x не определён».

Ответ

let x = undefined

if (x == "number"){

console.log("x - это число");

} else if(x == "string"){

console.log("x - это строка");

} else if(x == "boolean"){

console.log("x -это логический тип");

} else {

console.log("тип x - не определён");

}

console.log(typeof x)

При возникновении проблем по ходу решения вы всегда можете обратиться к ментору в Slack.

Удачи!

### [Операторы присваивания](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#assignment_operators)

### Возведение в степень (\*\*=)

Оператор присваивания возведения в степень ( \*\*=) возводит значение переменной в степень правого операнда.

### [Арифметические операторы](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#arithmetic_operators)

### Возведение в степень (\*\*)

Оператор возведения в степень ( \*\*) возвращает результат возведения первого операнда в степень второго операнда. Он эквивалентен Math.pow, за исключением того, что он также принимает BigInts в качестве операндов.

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя** | **Сокращённый оператор** | **Смысл сокращённо** | **Смысл развёрнуто** |
| Присваивание | x = y | x = y | Задаёт переменной x значение y. |
| Присваивание со сложением | x += y | x = x + y | Задаёт переменной х (слева) значение равное предыдущему значению x плюс значение y. |
| Присваивание с вычитанием | x -= y | x = x - y | Задаёт переменной х (слева) значение равное предыдущему значению x минус значение y. |
| Присваивание с умножением | x \*= y | x = x \* y | Задаёт переменной х (слева) значение равное предыдущему значению x умножить на значение y. |
| Присваивание с делением | x /= y | x = x / y | Задаёт переменной х (слева) значение равное предыдущему значению x делённое на значение y. |
| Присваивание по модулю | x %= y | x = x % y | Задаёт переменной х (слева) значение равное остатку от деления предыдущего значения x на значение y. |

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| Равно (==) | Возвращает true, если операнды равны. Оператор сначала приводит операнды к одному типу, и затем применяет строгое сравнение. |
| Не равно (!=) | Возвращает true, если операнды не равны. Аналогично оператору равенства, перед сравнением приводит операнды к одному типу. |
| Строго равно (===) | Возвращает true, если операнды строго равны и имеют одинаковый тип. |
| Строго не равно(!==) | Возвращает true, если операнды не равны и/или имеют разный тип. |
| Больше (>) | Возвращает true, если операнд слева больше операнда справа. |
| Больше или равно (>=) | Возвращает true, если операнд слева больше или равен операнду справа. |
| Меньше (<) | Возвращает true, если операнд слева меньше операнда справа. |
| Меньше или равно (<=) | Возвращает true, если операнд слева меньше или равен операнду справа. |

Операторы Сравнения

Оператор Название Пример

> Больше 3 > 1 -> true 3 > 5 -> false;

< Меньше 6 < 9 -> true 6 > 15 -> false;

>= Больше или равно 3 >= 3 -> true 3 >= 10 -> false ;

<= Меньше или равно 4 <= 4 -> true 4 <= 1 -> false

== Равенство 10 == 10 -> true 10 == “10” -> true 4 == 6 -> false;

=== Строгое равенство 10 === “10” -> false 3 === 3 -> true;

!= Неравенство 4 != 5 -> true 6 != 6 -> false 6 != “6” -> false

!== Строгое неравенство 3 !== 8 -> true 3 !== 3 -> false 3 !== “3” -> true

Логическое ИЛИ Логическое ИЛИ (||) вернёт true, если хотя бы одно из значений в логическом выражении равно

true true || true -> true;

false || true -> true ;

true || false -> true;

false || false -> false

Логическое И Логическое И (&&) вернет true, только если все значения в логическом выражении равны true .

true && true -> true;

false && true -> false;

true && false -> false;

false && false -> false

Логические операторы Логическое НЕ Логическое НЕ (!) применяется только с одним аргументом и возвращает противоположное ему значение

!true -> false

!false -> true

if (условие1)

инструкция1

else if (условие2)

инструкция2

else if (условие3)

инструкция3

...

else

инструкция

let x;

console.log(x=10);

10

Let a, b; - инструкция не возвращает значение

b = a = 10; выражение возвращает значение

10

Если я правильно понял, то :  
let x = 10 это “statement” а не выражение. Поэтому не возвращает значение.x = 10; без let это  выражение. И поэтому оно возвращает значение )

Это создает переменную в глобальной области видимости (Global scope) с именем x со значением 10. Кроме того, это выражение, которое возвращает значение 10. Это полезно для того, чтобы иметь возможность делать такие вещи, как var x = y = 10; который устанавливает для x и y значение 10

случай let х = 10:

Это создает переменную в текущей области видимости, которая является глобальной областью видимости, с именем x и значением 10. Поскольку она создается с использованием синтаксиса var/let/const, ее нельзя вычислить как выражен

Давайте добавим всем пользователям свойство *city*. =>

let users = **[**

**{** name: 'Steve', age: 18 **}**,

**{** name: 'John', age: 12 **}**,

**{** name: 'Liam', age: 16 **}**,

**{** name: 'Mike', age: 20 **}**,

**{** name: 'Jacob', age: 21 **}**

**]**

users.forEach**(**user => **{**

user.city = 'New York';

**})**;

**Функция** — это блок программного кода, который определяется один раз и может выполняться (вызываться) многократно.

Функция в JavaScript — это объект, который может быть вызван как **подпрограмма**.

**ВАЖНО!**Все функции в JavaScript — это объекты!

О. Уже теплее.  Спасибо. Т.е. я правильно понимаю, я как разработчик могу пользователю на его заданный вопрос ответить  значением, которое вернула функция ( своего рода такое себе значение-ответ), после проведения вычислений внутри себя. Поскольку функция подпрограмма, она возвращает значение (ответ) в основную программу, с помощью которой я общаюсь с пользователем и принимаю от него вопросы , и даю ему ответы значениями, которые возвращаются функцией в основную программу. Попросту говоря, я дал  руль другому человеку, он какое-то время порулил и должен мне этот руль вернуть после проведения каких-то виражей. Возврат руля это есть значение или код, ответ, результат, который выполнила функция и вернула мне, как владельцу? Я все верно понимаю? (отредактировано)

https://ca.slack-edge.com/TNU0H0XHB-U01DN29N76U-f19b1145187d-48

[**Daria Korotkova**](https://app.slack.com/team/U01DN29N76U)  [3 мин назад](https://sffrontenddev.slack.com/archives/C019CL21NFL/p1660083199619169?thread_ts=1660073126.625669&cid=C019CL21NFL)

[@Сергей](https://sffrontenddev.slack.com/team/U03BWC0NB6E) да все верно,отличная аналогия с рулем:)

https://ca.slack-edge.com/TNU0H0XHB-U03BWC0NB6E-e7e0f6312e37-48

[**Сергей**](https://app.slack.com/team/U03BWC0NB6E)  [2 мин назад](https://sffrontenddev.slack.com/archives/C019CL21NFL/p1660083285040509?thread_ts=1660073126.625669&cid=C019CL21NFL)

Спасибо!!! Наконец-то расчехлил этот вопрос:легкая_улыбка:

Функцией называется выделенная последовательность инструкций, предназначенных для решения определенной задачи. Ранее мы уже использовали библиотечные функции ввода-вывода printf() и scanf(), в данной лабораторной работе познакомимся с правилами создания своих (пользовательских) функций.

Есть несколько причин для использования пользовательских функций, во-первых, программа приобретает некоторую структуру и, тем самым, становится более понятной и упорядоченной, во-вторых, исключаются повторы похожих участков текста, то есть текст программы оптимизируется.

Функция может многократно вызываться из различных частей программы, в

общем случае она выполняет следующие действия:

· получает параметры;

· выполняет инструкции, согласно заложенному алгоритму;

· может возвращать результат в вызывающую программу.

Есть локальные, есть глобальные.

Возвращает значение функции - это значит выдает результат вычисления функции.

Возвращает значение функции - это значит что функцию можно использовать как переменную в операторе присвоения или в другой функции (например printf). Например. Есть функция delat\_chto-to( ); если в ней написано например return 2; то можно написать A=3+delat\_chto-to( ); и получится что A=5; Чаще функция что-то делает а не вычисляет, например переставляет местами элементы массива. Тогда в return можно записать например коды ошибок. 0 если все хорошо, 1 если нет. Короче говоря, с помощью оператора return мы присваиваем какое-то значение идентификатору ф-ии, (то есть названию ф-ии) после чего название ф-ии можно использовать как обычную переменную. Таким образом, ф-я выполняется, делает то, что ей положено, и одновременно с помощью return названию ф-ии присваивается какое-то значение. Иногда єто бывает полезно. Если присваивать значение особо не нужно, то используют значения для кодов ошибки.

Для чего нужны функции в js ?Для того чтобы иметь возможность создавать (абстpакцию) удобное понятие для рассуждения и построение своей логики при написании программ и алгоритмов. Функции в JS прячут в себе определенную функциональность (программу) для того чтобы сократить код и упростить понимание и рассуждения в процессе написания своей программы.

В JS функции являются объектом первого класса JS это язык с \_очень\_ высоким уровнем абстракций а это значит что функции являются не только инструментом который изменяет данные но и сами функции являться объектами данных которые могут изменять другие инструменты или же сами функции. Функции в JS прячут в себе определенную функциональность (программу) для того чтобы сократить код и упростить понимание и рассуждения в процессе написания своей программы.

Просмотрев первые видео, я могу предположить, что функция возвращает значение для того, чтобы программа просто начала работать, скажем, для автоматического включения программы, даже если в функции не прописаны еще условия с помощью if, else и мы пока  не видим результатов работы функции, но программа уже запущена благодаря возврату значения, которое возвращает функция или переменная. Поправьте меня пожалуйста если я, что-то не правильно говорю. За ранее спасибо. (отредактировано) ???

Дело в том, что return означает не возврат информации, а возврат управления.

При вызове функции текущий код становится на паузу и передаёт управление в функцию. Функция делает свои дела и **возвращает управление** в то место, где происходил вызов. И заодно может отправить какую-то информацию туда.

Что за управление? Управление той штукой, которая исполняет код. Грубо говоря — управление компьютером. Код передаёт функции пульт управления компьютером, а функция потом возвращает его обратно.

**const** sum **=** (a, b) **=>** {

**const** answer **=** a **+** b;

**return** answer;

}

Мы привыкли читать подобный код как «функция возвращает значение answer», но правильнее будет читать «функция возвращает управление в место вызова и отправляет туда значение answer».

Функция всегда создается с какой-то целью, например, рассчитать какое-то значение или отформатировать строку определенным образом, для этого и нужен возврат значения - чтобы передать в среду выполнения результат расчетов и часть программы за пределами функции могла их использовать.  
На курсе есть раздел по функциям в Javascript, там это всё объясняется более подробно и с примерами) – Ю. Токаревская

**Задание 6.**

Дан массив. Проверить, одинаковые ли элементы в массиве и вывести результат true или false в консоль. Речь идёт не о двух рядом стоящих одинаковых элементах, а обо всех. Проверить, все ли элементы в массиве одинаковые.

let numArr = [3, 55, 8, 7, 77, 3, 20];

let check = numArr[0];

numArr.forEach((item, index, numArr) => {

if(item == check) {

console.log(`true`);

} else {

console.log(`false`);

}

})

let arr = [3, 77, 8, 7, 77, 20, 3];

let element = arr[0];

for (item of arr){

if(item !== element){

console.log("false");

} else{

console.log("true");

}

}

**Задание 7.**

Дан массив. Нужно вывести в консоль количество чётных и нечётных элементов в массиве. Если в массиве есть нулевой элемент, то он учитывается и выводится отдельно.

При выполнении задания необходимо учесть, что массив может содержать не только числа, но и, например, знаки, null и так далее.

let diffArr = [0, 7, null, 1, 3, 4, 8, "hello"];

let sumX = 0;

let sumOdd = 0;

let sumPar = 0;

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

if (diffArr[i] == 0 ) {

sumX +=1;

} else if (typeof (diffArr[i]) !== "number" || typeof (diffArr[i]) == "NaN") {

console.log(`${diffArr[i]} - это не число`);

} else if (diffArr[i] %2 == 0) {

sumPar +=1;

} else {

sumOdd +=1;

}

}

console.log(`${sumX} - нулей`);

console.log(`${sumPar} - не чётных числа`);

console.log(`${sumOdd} - чётных числа`);

При возникновении проблем по ходу решения вы всегда можете обратиться к ментору в Slack.

Удачи!

for (let i = 0; i < diffArr.length; i++)

|  |  |  |
| --- | --- | --- |
| *начало* | let i = 0 | Выполняется один раз при входе в цикл |
| *условие* | i < 3 | Проверяется *перед* каждой итерацией цикла. Если оно вычислится в false, цикл остановится. |
| *тело* | alert(i) | Выполняется снова и снова, пока условие вычисляется в true. |
| *шаг* | i++ | Выполняется *после* тела цикла на каждой итерации *перед* проверкой условия. |

Выполнить начало

→ (Если условие == true → Выполнить тело, Выполнить шаг)

→ (Если условие == true → Выполнить тело, Выполнить шаг)

→ (Если условие == true → Выполнить тело, Выполнить шаг)

→ ...

// for (let i = 0; i < 3; i++) {console.log (i)}

// Выполнить начало

let i = 0;

// Если условие == true → Выполнить тело, Выполнить шаг

if (i < 3) { console.log (i); i++ }

// Если условие == true → Выполнить тело, Выполнить шаг

if (i < 3) { console.log (i); i++ }

// Если условие == true → Выполнить тело, Выполнить шаг

if (i < 3) { console.log (i); i++ }

// ...конец, потому что теперь i == 3

Проверка четных и нечетных чисел

Там все просто, модуло в js это %  
К примеру у нас есть натуральные числа : 2 и 3  
Надо проверить какое четное какое нет.  
Если поделить 2 % 2 то будет 1 и 0 в остатке  
Если поделить 3 % 2 то будет 1 и 1 в остатке  
По остатку, деления числа на 2, можно узнать чет это или нечет.

Задуманное число при деление на 2 дает остаток 1,а при делении на 3-остаток 2.

Что значит "функция возвращает значение", ты понимаешь?

Например,  
x=Math.cos(0)  
- функция Math.cos возвращает значение косинуса аргумента, в данном случае 1, после чего оператор = записывает это значение в x.  
Но оператор = тоже возвращает значение, и мы это можем использовать, например, так:  
alert(x=Math.cos(x)) - значение, возвращенное функцией, будет записано в x, а затем значение, возвращенное оператором = - то же самое, в этом случае - в функцию alert.  
Другой пример:  
x=y=Math.sin(1)  
- разбери сам.  
Еще пример:  
y=(x>1);  
if(y)alert("Да");

Это значит, что результатом выполнения оператора является некоторое значение. Например, оператор сложения возвращает сумму стоящих справа и слева от него операндов. А вот какой-нибудь управляющий оператор вроде try никаких значений не возвращает. Смекаешь?

"Функция возвращает значение F(x)" означает, что она производит отображение X --> Y, где x принадлежит множеству X, а y = F(x) принадлежит множеству Y. Само слово "возвращает" восходит к механизму вызова подпрограмм, а именно: существует момент, когда некоторая функция вызыется (происходит переход на её код) и когда происходит возврат из этой функции (всегда в вызывающую), значение функции становится известно в вызывающей функции только когда произойдёт, собственно, возврат из неё, поэтому и говорят "возвращаемое значение", "функция возвращает".

function SomeFunc: Integer;  
begin  
 Result := 1  
end;  
  
N := SomeFunc();  
  
Так вот, функция SomeFunc вовзращает значение 1 вызывающему ее коду.



Параметрами являются **a** и **b**. Параметры перечисляются через запятую.

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

**const** result = a + b;

console.log(result); };

При вызове функции ей передаётся список значений, называемых**аргументами** функции. Например, 1 и 2 являются аргументами функции sum при вызове:

sum(**1**,**2**)

Аргументы перечисляются через запятую. После последнего аргумента ставить запятую **НЕ** нужно.

В JavaScript нет требования, чтобы количество переданных аргументов совпадало с количеством параметров. Валидны будут эти четыре вызова функции, но практически работать все будут по-разному:

**sum(1,2);**

**sum();** // не передаём аргументы

**sum(1);** // передаём меньше заданного количества параметров

**sum(1,2,3);** // передаём больше заданного количества параметров

Если передано больше аргументов, чем объявлено параметров, — sum(1,2,3) — то лишние переданные значения никак не повлияют на выполнение функции, и она отработает как sum(1,2);.

Если передано меньше аргументов, чем объявлено параметров, параметры, для которых не было передано аргументов, получат значение undefined.

То есть, вызвав sum(1), получим:

a = 1, b = undefined

result = 1 + undefined

У каждой функции есть read-only (то есть неизменяемое) свойство length, которое определяется в момент создания функции и содержит число параметров функции.

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

**const** result = a + b;

console.log(result);

}

console.log(sum.length);

**2**

Это число называется **арность**(*arity*) — количество параметров у функции.

Для доступа к аргументам может также использоваться псевдомассив (объект, у которого числовые имена — ключи — свойств и есть свойство length) аргументов arguments.

Он содержит список аргументов по номерам: arguments[0], arguments[1] и так далее, а также свойство length.

На практике его используют тогда, когда задано больше аргументов, чем параметров.

Важно помнить, что, если задан параметр, но в функцию не передан аргумент, по умолчанию он равен undefined.

**function** log(a) {

console.log(a);

}

log(2); // аргумент передан, выведется 2

log(); // аргумент не передан, выведется undefined

Если значение undefined может критично сказаться на работе функции, можно задать параметры по умолчанию (или дефолтные параметры).

Это делается путём присваивания значения при объявлении параметра.

При этом, если аргумент передан, дефолтное значение гарантированно будет отброшено:

**function** log(a = **0**) {

console.log(a);

}

log(2); // аргумент передан, выведется 2

log(); // аргумент не передан, выведется дефолтное 0

**Функция и процедура**

И процедура, и функция — набор инструкций (подпрограмма), который может быть выполнен в определенном порядке.

Отличие заключается в том, что функция всегда должна возвращать значение (более строгое определение требует также, чтобы у функции всегда были входные данные).

По умолчанию возвращаемое значение — undefined.

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

a + b;

};

console.log(sum(**1**, **2**));

**undefined**

Одной из особенностей JavaScript, которая делает его столь удобным для функционального программирования, является то, что он может принимать функции высшего порядка. Функция высшего порядка — это функция, которая может принимать другую функцию в качестве аргумента или возвращать другую функцию в качестве результата.

**Функции высшего порядка(значения возвращаются в функцию в первую функцию в виде названия функции, в которой выполняется действие)**

**Декларативный** же подход отвечает на тот же самый вопрос гораздо короче — «Ленина, 52».

**function exemple(func){**

**func();**

**return function(){**

**console.log('hello')**

**}**

**}**

**function argFunc() {**

**console.log('world')**

**}**

**const resultFunk = exemple(argFunc);**

**resultFunk();**

**function** example(func){

// вызываем функцию, которую приняли как аргумент

func();

// возвращаем функцию

**return** **function**(){

console.log('internal function')

}

}

// функция, которую будем передавать как аргумент в функцию example

**function** argFunc(){

console.log('function from arguments')

}

// в переменную resultFunc записываем результат выполнения функции example

// функция, которая передаётся как аргумент, должна быть без круглых скобок

**const** resultFunc = example(argFunc);

// после вызова example() внутри неё будет вызвана функция argFunc, которую мы передали как аргумент, и в консоли будет текст: function from arguments

// вызываем функцию

resultFunc(); // 'internal function

**Задание 3. Модуль 6**

Написать функцию, которая принимает число как аргумент и возвращает функцию, которая также принимает число как аргумент и возвращает сумму этих двух чисел. Выведите в консоль результат.

<https://egnat.github.io/module5_homework-Nalivaiko-Sergey/>

Последнее [развертывание](https://github.com/Egnat/module5_homework-Nalivaiko-Sergey/actions/runs/2783773440) на @github-страницы [github-pages](https://github.com/github-pages) 3 minutes ago

[Посетите сайт](https://egnat.github.io/module5_homework-Nalivaiko-Sergey/)

Упражнения отЮ.Токаревской

// Упражнение 8. Дан массив элементов arr. Нужно создать функцию unique(arr), которая вернёт массив уникальных, не повторяющихся значений массива arr

let commonArr = [13, 13, 20, 45, 66, 20, 18, 15, 20];

function unique(arr) {

return Array.from(new Set(arr));

}

console.log(unique(commonArr));

(? unique)" - (? уникальных)".

**Задание 3. Module 6**

Написать функцию, которая принимает число как аргумент и возвращает функцию, которая также принимает число как аргумент и возвращает сумму этих двух чисел. Выведите в консоль результат.

Удачи!

**Задание 4. Module 6**

Напишите функцию, которая принимает два числа. Каждую секунду необходимо выводить в консоль, начиная от первого и заканчивая вторым. Используйте setInterval.

Например, пользователь ввёл числа 5 и 15. Каждую секунду в консоль должно печататься число, начиная с 5 и заканчивая 15 (всего 11 чисел: 5 6 7 8 9 10 11 12 13 14 15).

Удачи!

ptr = setInterval(func, timeout);

Разница в них такая, что setTimeout() отрабатывает ровно один раз, т.е. вызывает через указанный интервал указанную функцию и останавливает свою работу. Вторая функция же, setInterval() принимает точно такие же параметры, но вызвав функцию однажды, она продолжает делать это снова и снова через одинаковые промежутки времени. Остановить её работу можно только с помощью функции clearInterval(ptr), где ptr — ссылка на объект, полученная как результат выполнения функции setInterval().

**Задание 5. Module 6**

Напишите функцию, которая принимает два натуральных числа ***x*** и **n** и возвращает ***x*** в степени **n**. Иначе говоря, умножает ***x*** на себя ***n*** раз и возвращает результат.

Используйте Arrow Function синтаксис.

Протестируйте функцию на любых значениях и выведите результат в консоль.

Удачи!

const getdegree = x => {

x();

return (x)

console.log();

}

const argX = () => {

console.log(2\*3);

}

const resultArgX = getdegree(argX)

resultArgX();

const getdegree = sum => {

sum();

return sum = () => {

console.log(3);

}

}

const argSum = () => {

console.log(3\*2);

}

const resultArgSum = getdegree(argSum)

resultArgSum();

const getdegree = sum => {

//sum();

return sum = () => {

console.log(3);

}

}

const argSum = () => {

console.log(3\*2);

}

const resultArgSum = getdegree(argSum)

resultArgSum();

This называют **контекстом**. *This* является объектом, который ссылается на саму функцию. То есть, таким образом, мы записываем в функцию свойства, которые затем можно передать новому, созданному через *new*, экземпляру.

1. Ключевое слово *this* указывает на текущий контекст вызова функции. Контекст есть только у функций (в том числе, методов объекта, функций–конструкторов).
2. Контекст нужен для доступа к информации внутри объекта. Если внутри функции есть обращение к this — ожидается, что функция вызвана в контексте какого–то объекта.
3. Контекст не статичен — он определяется в *момент выполнения функции*. Его называют **динамическим**.
4. Ключевое слово *this* во время вызова функции хранит ссылку на объект, для которого эта функция была вызвана в качестве метода. Так можно получить доступ к другим свойствам этого объекта.
5. В случае, когда функция вызывается сама по себе, а не как метод объекта, тогда есть два варианта поведения:

* Ключевое слово this будет ссылаться на глобальный объект *window*
* Если используется строгий режим ('use strict') – this будет иметь значение *undefined*

Модуль 7

**Задание 1.**

Написать, функцию, которая принимает в качестве аргумента объект и выводит в консоль все ключи и значения только собственных свойств. Данная функция не должна возвращать значение.

**Задание 2.**

Написать функцию, которая принимает в качестве аргументов строку и объект, а затем проверяет есть ли у переданного объекта свойство с данным именем. Функция должна возвращать true или false.

function arg (obj) {

const newObj = {};

for (let prop in obj) {

if (obj.hasOwnProperty(prop)) {

newObj[prop] = obj[prop];

}

}

return true

}

const a = {b: 'hello'};

const b = arg(a);

console.log(b);

console.log(a, b)

///////

let a;

function arg (obj = {a: 'hello'}) {

for (let key in obj) {

if (obj.hasOwnProperty(key)) {

console.log(key);

if(key !== 'hello') {

console.log(true);

}else {

console.log(false);

}

}

}

return true;

}

arg( );

**Задание 3.**

Написать функцию, которая создает пустой объект, но без прототипа.

Удачи!

**Задание 4.**

Реализуйте следующее консольное приложение подобно примеру, который разбирался в видео. Реализуйте его на прототипах.

Определите иерархию электроприборов. Включите некоторые в розетку. Посчитайте потребляемую мощность (передайте аргумент).

Таких приборов должно быть как минимум два (например, настольная лампа и компьютер). Выбрав прибор, подумайте, какими свойствами он обладает.

**План:**

1. Определите родительскую функцию с методами, которые включают/выключают прибор из розетки.
2. Создайте делегирующую связь [[Prototype]] для двух конкретных приборов.
3. У каждого из приборов должны быть собственные свойства и, желательно, методы, отличные от родительских методов.
4. Создайте экземпляры каждого прибора.
5. Выведите в консоль и посмотрите на результаты работы, можете гордиться собой :)

**function Device(name) {**

**this.electrical = 'network',**

**this.name = name**

**}**

**Device.prototype.getElectricity = function(electricity) {**

**console.log(`The TV consumes ${electricity} watts of electricity`)**

**}**

**function ElectricDevice(name, light) {**

**this.name = name,**

**this.light = light**

**}**

**ElectricDevice.prototype = new Device()**

**ElectricDevice.prototype.getElectricity = function(electricity) {**

**console.log(`The device consumes ${electricity} watts of electricity`)**

**}**

**const lamp = new ElectricDevice('lamp', 'switch');**

**const comp = new ElectricDevice('comp');**

**const tv = new Device('tv');**

**lamp.getElectricity(40)**

**comp.getElectricity(60)**

**tv.getElectricity(100)**

**///////////**

**function ElectricalDevices(electricity) {**

**this.consumes = electricity,**

**this.electrical = 'network'**

**}**

**const tableLamp = new ElectricalDevices(40);**

**const comp = new ElectricalDevices(60);**

**const tv = new ElectricalDevices(100);**

**console.log(tableLamp, comp, tv)**

**Задание 5.**

Переписать консольное приложение из предыдущего юнита на классы.

* Имена классов, свойств и методов должны быть информативными;
* Соблюдать best practices;
* Использовать синтаксис ES6.

class Parent {

constructor(ownDevice) {

this.ownDevice = ownDevice || 'on average 50 vt'

this.fasFlat = true;

}

getInfo() {

return 'Lamp consume ' + this.ownDevice;

return 'Comp consume ' + this.ownDevice;

}

}

class Child extends Parent {

constructor(isElectrical, light, ownDevice) {

super(ownDevice);

this.isElectrical = isElectrical;

this.light = isElectrical ? light : ownDevice;

}

getInfo() {

if (this.isElectrical) {

return 'Lamp consume ' + this.light

} else {

return super.getInfo()

}

}

}

class Car {

}

console.log(electrical instanceof Car)

const electrical = new Child(true, 'min 40 vt');

console.log(electrical.getInfo())

const gas = new Child(false, 'max 90 vt');

console.log(gas.getInfo())

console.log(electrical instanceof Child)

console.log(electrical instanceof Parent)

//////////////////////////////////////////////

// на проверку

class Parent {

constructor(ownDevice, ownDevices) {

this.ownDevice = ownDevice || 'on average 50 vt'

this.ownDevices = ownDevices || 'on average 50 vt'

this.fasFlat = true;

}

getInfo() {

return 'Table lamp consume ' + this.ownDevice;

}

getInf() {

return 'Personal comp consume ' + this.ownDevices;

}

}

class Child extends Parent {

constructor(isElectrical, light, ownDevice) {

super(ownDevice);

this.isElectrical = isElectrical;

this.light = isElectrical ? light : ownDevice;

}

getInfo() {

if (this.isElectrical) {

return 'Table lamp consume ' + this.light

} else {

return super.getInfo()

}

}

}

class Child1 extends Parent {

constructor( isElectric, lighting, ownDevices) {

super(ownDevices);

this.isElectric = isElectric;

this.lighting = isElectric ? lighting : ownDevices;

}

getInf(){

if (this.isElectric) {

return 'Personal comp consume ' + this.lighting

} else {

return super.getInf()

}

}

}

const electrical = new Child(true, 'min 40 vt');

console.log(electrical.getInfo())

const gas = new Child(false, 'max 90 vt');

console.log(gas.getInfo(), gas.getInf())

const electric = new Child1(true, 'max 60 vt');

console.log(electric.getInf())

console.log(electrical instanceof Child)

console.log(electrical instanceof Parent)

console.log(electric instanceof Child1)

console.log(electric instanceof Parent)

class Car {

}

console.log(electrical instanceof Car)

console.log(electric instanceof Car)

//constructor

function Parent(ownDevice) {

this.ownDevice = ownDevice;

this.fasFlat = true;

}

Parent.prototype.getInfo() {

console.log(`Lamp consume` ${this.ownDevice});

}

let parent = new Parent('on average 50 vt ');

//

class Electrical {

static electric() {

console.log('divaces')

}

divace() {

console.log('vt');

}

}

class Electricity extends Electrical {

static awesome() {

super.cool();

console.log('awesome')

}

great() {

super.cool();

console.log('great')

}

}

Electrical.electric();

Electricity.electric();

Electricity.electric();

/\*const instance = new Electricity();

instance.cool();

instance.divace();

instance.electric();\*/

МОДУЛЬ 8

DOM, BOM, Js

Итак, продолжим, что значит Триггерить? Этот термин был заимствован из английского языка «trigger«, и переводится, как «курок», «зацепка». Триггерить — означает, быть зацепленным чем-то, то есть обратить на что-то своё внимание, на то, что вас цепляет. Триггер — (англ. trigger) спусковое устройство (Спусковая схема), которое может сколь угодно долго находиться в одном из двух (реже многих) состояний устойчивого равновесия и скачкообразно переключаться из одного состояния в другое под действием…

**Поиск, dropzone**

**Поиск**

getElementById

, draggable

Placed in: Поиск

getElementsByTagName

, draggable

Placed in: Поиск

getElementsByClassName

, draggable

Placed in: Поиск

querySelector

, draggable

Placed in: Поиск

querySelectorAll

, draggable

Placed in: Поиск

Items placed here: getElementById, getElementsByTagName, getElementsByClassName, querySelector, querySelectorAll

**Изменение, dropzone**

**Изменение**

setAttribute

, draggable

Placed in: Изменение

style.property

, draggable

Placed in: Изменение

innerHTML

, draggable

Placed in: Изменение

Items placed here: setAttribute, style.property, innerHTML

**Добавление, dropzone**

**Добавление**

innerHTML

, draggable

Placed in: Добавление

createElement

, draggable

Placed in: Добавление

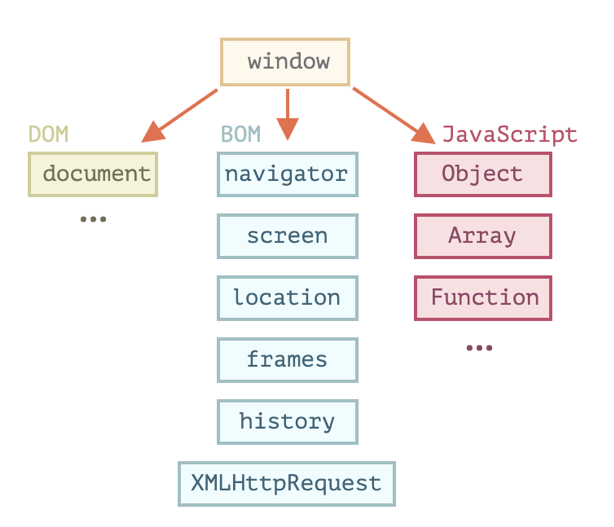
insertAdjacentHTML

, draggable

Placed in: Добавление

В данном курсе мы работаем с браузером, который выступает так называемым **окружением**. Окружение предоставляет нам объекты и функции для работы с ним. К примеру, мы уже использовали метод setTimeOut, предоставляемый браузером.

 Схематично браузерное окружение выглядит так:



Где window — глобальный объект, в котором хранится вся доступная функциональность. Он используется, когда разработчик хочет использовать методы, предоставляемые браузером.

Window выступает в двух ролях:

* 1. Как глобальный объект для JS-кода (подробнее о глобальном объекте [тут](https://learn.javascript.ru/global-object));
  2. Как окно браузера.

Внутри window мы видим DOM, BOM и JS.

1

**DOM**(document object model) — это объектная модель документа. Является Web API, так как предоставляется браузером.

DOM поддерживает объектно-ориентированное представление веб-страницы, делая возможным её изменение при помощи, например, JavaScript. Работе с DOM посвящена основная часть этого модуля, подробности вас ждут в следующих видео.

2

**BOM** (browser object model) — это браузерное окружение. Оно предоставляет функционал (объекты и функции), который может использовать JavaScript, например, BOM включает в себя такие часто используемые вещи, как:

* + **Location** — возвращает объект Location с информацией о текущем расположении документа;
  + **History** — предоставляет интерфейс для манипулирования историей сессии браузера (например, страницы, посещённые в текущей вкладке);
  + **XMLHttpRequest** — для создания запросов к серверу.

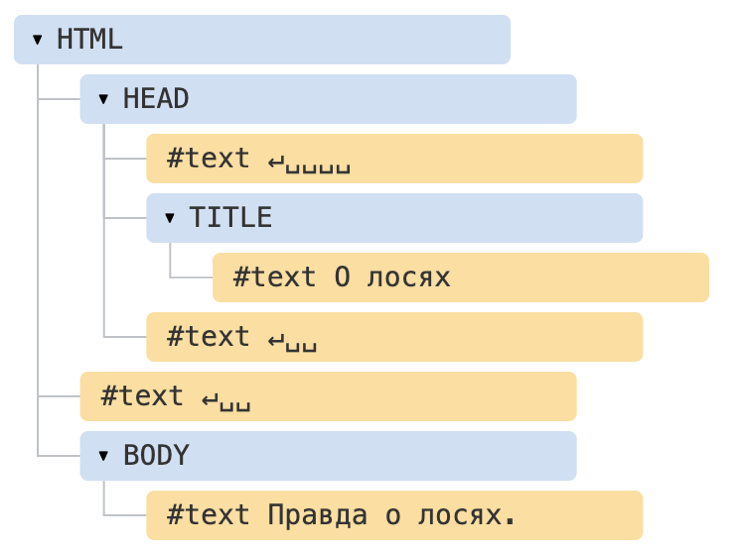
3

С **JS** в этом контексте вы уже знакомы — это все [глобальные объекты](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects), о которых мы много говорили в предыдущих модулях.

Узлом называют HTML-элемент.

Узлы бывают разных типов:

* + Узлы (*HEAD*),
  + Текст (#*text*),
  + Комментарий (#*comment*),
  + *Document* (входная точка в *DOM*. По сути — узел).



Чтобы узнать тип узла, можно воспользоваться свойством elem.nodeType. У каждого объекта есть свойства, методы и события.

* **Свойства** — то, что характеризует объект (узел);
* **Методы** — то, что объект способен делать сам, без внешнего воздействия;
* **События** — влияние на объект извне.
* **ParentNodes**
* Кроме деления по типам, узлы делятся на **дочерние** и **родительские**. Соответственно, для обращения к элементам, вложенных в *DOM-*узел, существует два типа навигационных ссылок:
* **Для всех узлов**:
* *parentNode*
* **Только для узлов-элементов**:
* *parentElement*

**ChildNodes**

Дочерние узлы делятся ещё на два типа:

**Дочерние узлы**(*childNodes*) — элементы, которые лежат непосредственно внутри того элемента, к которому обращаются (неглубокая вложенность): body для *HTML*.

**Потомки** (*children*) — все элементы, которые лежат внутри тега, включая вложенные в другие теги: body, div, ul, li, b для *HTML.*

**Для всех узлов**:

*childNode, firstChild, lastChild*

**Только для узлов-элементов**:

*children, firstElementChild, lastElementChild*

Родительские узлы определяются однозначно. Обратиться к родительскому свойству можно через *parentElement* или *parentNode*. Они оба возвращают родителя.

Но есть небольшой нюанс:

* + *parentElement* вернёт узел-элемент,
  + *parentNode* — любого родителя.

**Пример**:

document.documentElement.parentNode === document

document.documentElement.parentElement === null

**ВАЖНО!** *document*— не узел, а объект.

Для всех остальных узлов результат одинаковый.

Для всех остальных узлов результат одинаковый.

**firstChild, lastChild**

Свойства *firstChild* и *lastChild* обеспечивают быстрый доступ к первому и последнему дочернему элементу из списка *childNodes*.

Проверить, существуют ли вообще дочерние узлы, можно с помощью функции elem.hasChildNodes().

*childNodes* — это псевдомассив, который можно перебрать с помощью for ... of.

**ВАЖНО!**Цикл *for ... in* не подходит для перебора.

Методы массивов для childNodes НЕ работают, но его можно преобразовать в массив, используя методы [built-in object Array](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/from), а затем использовать необходимые методы. Например, Map или forEach.

**ВАЖНО!**Изменить DOM, оперируя элементами childNodes, НЕЛЬЗЯ. Для этого существуют специальные методы, о которых вы узнаете далее.

## ****Siblings****

**Siblings**(соседи) — узлы, у которых один и тот же родитель.

**Правый** (следующий) узел — nextSibling.

**Левый** (предыдущий) узел — previousSibling.

**Родитель**— parentNode.

Например, теги head и body — соседи.

**Для всех узлов**:

previousSibling, nextSibling

**Только для узлов-элементов**:

previousElementSibling, nextElementSibling

**Поиск элемента в DOM**

**По *id***

document.getElementById(id)

Принимает *id* как строку. Возвращает элемент — объект.

2

**По названию тега**

document.getElementsByTagName(tagName)

Принимает название тега как строку. Возвращает массив элементов.

3

**По имени класса**

document.getElementsByClassName(className)

Принимает имя класса как строку. Возвращает массив элементов.

4

**По любому *CSS*-селектору**

document.querySelector(selector)

Принимает селектор как строку, первый символ или его отсутствие указывает на селектор (. — класс, # — id, отсутствие — имя тега). Возвращает первый элемент.

5

**По любому селектору**

document.querySelectorAll(selector)

Принимает селектор как строку, первый символ или его отсутствие указывает на селектор (. — класс, # — id, отсутствие — имя тега). Возвращает массив элементов.

**Изменение элементов**

**Вставка любого контента напрямую:**

elem.innerHTML = "content"

Прежде чем вставлять контент, необходимо найти элемент, используя специальные функции.

2

**Вставка/изменение атрибута:**

elem.setAttribute(attr, value)

Принимает название атрибута как строку и значение для этого атрибута тоже как строку.

Прежде чем добавлять атрибут, необходимо найти элемент, используя специальные функции.

3

**Добавление стиля:**

elem.style.property = "style"

elem.style.color = "red" - пример

elem.style = "font-sise: 20px; color = "red;" - пример

Прежде чем вставлять стиль, необходимо найти элемент, используя специальные функции.

**Добавление и удаление элементов**

**Добавление элемента**

document.createElement(elem)

Принимает название тега  и значение для этого атрибута как строку.

**Пример**:

**1. Создаём элемент**

const customDiv = document.createElement("div");

**2. Заполняем его**

customDiv.innerHTML = "<span id="custom">Привет!</span>";

**3. Ищем родительский элемент и добавляем созданный элемент в дерево DOM**

const parent = document.getElementById("id");

document.body.insertBefore(customDiv, parent);

Обратите внимание на innerHTML — это довольно интересный метод, с помощью которого можно удобно совершать манипуляции с DOM, и как вы могли заметить, он может являться альтернативой для document.createElement.

Давайте попробуем переписать пример, добавив динамичности в виде шаблонных строк.

**1. Описываем переменную, которая будет контентом новых тегов**

const content = "Привет!"

**2. Ищем родительский элемент и добавляем новые элементы в дерево DOM**

const parent = document.getElementById("id");

parent.innerHTML = `<div><span id="custom"> ${content}</span></div>`;

И ещё один вариант —  метод **insertAdjacentHTML**.

insertAdjacentHTML() разбирает указанный текст как HTML или XML и вставляет полученные узлы (nodes) в DOM-дерево в указанную позицию. Данная функция не переписывает имеющиеся элементы.

**Добавление элемента**

document.createElement(elem)

Принимает название тега  и значение для этого атрибута как строку.

**Пример**:

**1. Создаём элемент**

const customDiv = document.createElement("div");

**2. Заполняем его**

customDiv.innerHTML = "<span id="custom">Привет!</span>";

**3. Ищем родительский элемент и добавляем созданный элемент в дерево DOM**

const parent = document.getElementById("id");

document.body.insertBefore(customDiv, parent);

Обратите внимание на innerHTML — это довольно интересный метод, с помощью которого можно удобно совершать манипуляции с DOM, и как вы могли заметить, он может являться альтернативой для document.createElement.

Давайте попробуем переписать пример, добавив динамичности в виде шаблонных строк.

**1. Описываем переменную, которая будет контентом новых тегов**

const content = "Привет!"

**2. Ищем родительский элемент и добавляем новые элементы в дерево DOM**

const parent = document.getElementById("id");

parent.innerHTML = `<div><span id="custom"> ${content}</span></div>`;

И ещё один вариант —  метод **insertAdjacentHTML**.

insertAdjacentHTML() разбирает указанный текст как HTML или XML и вставляет полученные узлы (nodes) в DOM-дерево в указанную позицию. Данная функция не переписывает имеющиеся элементы.

Принимает как аргументы строки — position и контент.

Position определяет позицию добавляемого элемента относительно элемента, вызвавшего метод. Должно соответствовать одному из следующих значений (чувствительно к регистру):

**'beforebegin'**: до самого **element** (до открывающего тега);

**'afterbegin'**: сразу после открывающего тега **element** (перед первым потомком);

**'beforeend'**: сразу перед закрывающим тегом **element** (после последнего потомка);

**'afterend'**: после **element** (после закрывающего тега).

Ищем родительский элемент и добавляем новые элементы в дерево DOM:

const parent = document.getElementById("id");

parent.insertAdjacentHTML('afterend', `<div><span id="custom"> ${content}</span></div>`);

Документация [здесь](https://developer.mozilla.org/ru/docs/DOM/document.createElement), [здесь](https://developer.mozilla.org/ru/docs/Web/API/Element/innerHTML) и [здесь](https://developer.mozilla.org/ru/docs/Web/API/Element/insertAdjacentHTML).

2

**Удаление элемента**

elem.removeChild(child)

Принимает child — элемент, который должен быть удалён.

elem — родительский элемент, из которого будет удалён child.

**Пример**:

**1. Ищем элемент, который должен быть удалён и его родителя**

const removed = document.getElementById("custom");

const div = document.querySelector("div");

**2. Удаляем**

div.removeChild(removed)

Изучить больше примеров можно [здесь](http://shpargalkablog.ru/2013/08/appendchild-removechild-javascript.html).

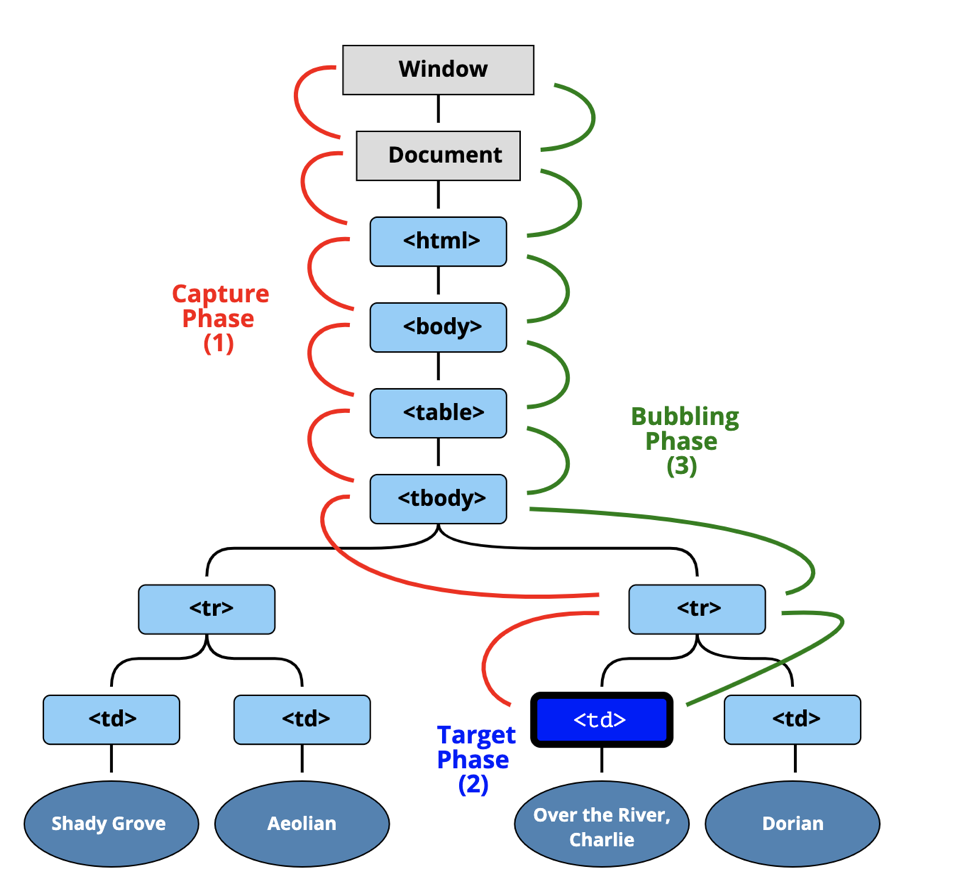
ФАЗЫ

Выделяют две фазы: **всплытие** (bubbling) и **погружение** (capturing).

Событие движется от корня документа к цели (фаза перехвата — Capturing), затем срабатывает для цели события (фаза цели — Target) и движется назад к корню документа (фаза всплытия — Bubbling).

По умолчанию, useCapture в addEventListener — это false, поэтому события будут всплывать.

**Как работает событие?**

Источник: [learn.javascript.ru](https://learn.javascript.ru/bubbling-and-capturing)

1. Когда браузер «триггерит» событие, элемент, на котором оно произошло, помечается как цель — это event.target.
2. После этого событие сначала двигается вниз от корня документа к event.target, по пути вызывая обработчики, поставленные через addEventListener(...., true), где true – это сокращение для {capture: true}. Это называется **погружением**.
3. Затем обработчики вызываются на event.target.
4. После этого событие двигается от целевого элемента вверх к корню документа, по пути вызывая обработчики, поставленные через on<event> и addEventListener без третьего аргумента или с третьим аргументом равным false. Это **фаза всплытия**.

**ВАЖНО!**Почти все события всплывают! Исключением, например, является событие focus.

Чаще всего, всплытие событий мешает. Чтобы предотвратить это, используют следующие функции:

1

**event.stopPropagation()**

События, которые идут за тем, в котором вызвана эта функция, на текущем элементе, отработают, всплытие остановится.

2

**event.stopImmediatePropagation()**

Все события, идущие за тем, которое вызывает эту функцию, не сработают, всплытие остановится.

3

**event.preventDefault()**

Он останавливает действия браузера по умолчанию. Аналогично, можно из обработчика, заданного через on<event> вернуть false.

**События мыши:**

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

**События на элементах управления:**

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

**Клавиатурные события:**

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

**События документа:**

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

**CSS events:**

* transitionend – когда CSS-анимация завершена.

Существует множество других событий. Мы подробно разберём их в последующих главах.

## [Обработчики событий](https://learn.javascript.ru/introduction-browser-events" \l "obrabotchiki-sobytiy)

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](https://learn.javascript.ru/introduction-browser-events" \l "ispolzovanie-atributa-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():

<script>

function countRabbits() {

for(let i=1; i<=3; i++) {

alert("Кролик номер " + i);

}

}

</script>

<input type="button" onclick="countRabbits()" value="Считать кроликов!">

**Удаление обработчика**

Существует еще один метод для работы с событиями, который называется removeEventListener. Он нужен для того, чтобы удалять обработчики событий.  Это может быть полезно, когда вы точно знаете, что событие больше не должно выполняться или существует условие для его отмены. В этом случае необходимо воспользоваться removeEventListener.

Метод имеет такой же синтаксис, как и addEventListener, и использовать они должны строго **одну и ту же функцию.**

element.removeEventListener(event, handler [, phase]);

Где:

**event** — тип события, которое нужно удалить,

**handler** — функция-обработчик, которую мы удаляем,

**phase** — фаза (обычно не используется, по умолчанию = false)

**ВЕРНО:**

<button onclick="click()" id="btn">Press</button>

<script>

**function** click1() {

alert('Click1');

};

btn.addEventListener("click", click1); // Click1

btn.removeEventListener("click", click1); //

</script>

**НЕВЕРНО:**

<button onclick="click()" id="btn">Press</button>

<script>

btn.addEventListener("click", **function**() {

alert('Click1');

};); // Click1

btn.removeEventListener("click", **function**() {

alert('Click1');

};); //

</script>

HTML-элемент <form> определяет форму для сбора информации, вводимой пользователем:

<form>

      <!-- элементы формы -->

</form>

**Элементы формы**— это различные типы полей для ввода информации, такие как текстовые поля, чекбоксы, радио-кнопки и так далее.

## ****Элемент <input>  и семантическая верстка****

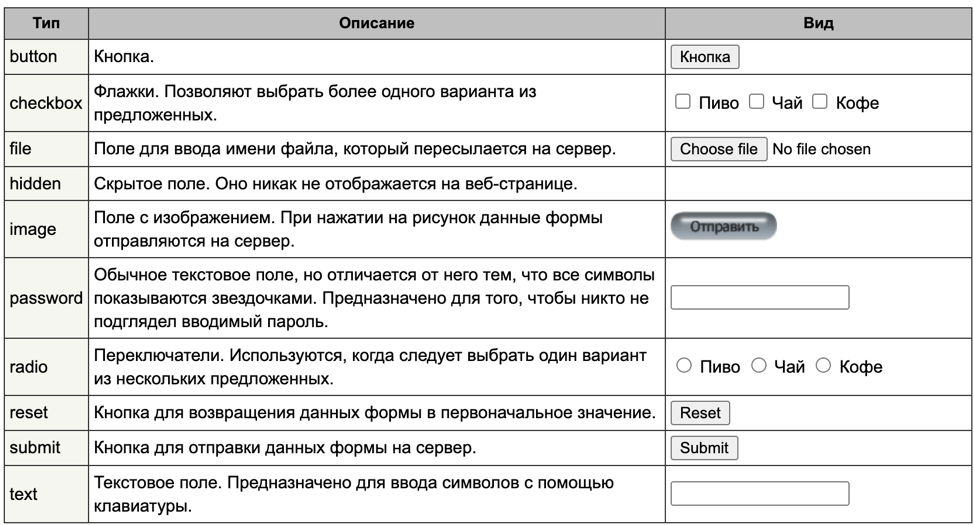
Элемент <input> может быть отображен различными путями, в зависимости от атрибута type.

<form>

     <input type="text" name="form\_input">

</form>

Вот некоторые из них:

Источник: [htmlbook.ru](http://htmlbook.ru/html/input/type)

Семантическая верстка подразумевает использование специальных HTML-элементов с определенной ролью —

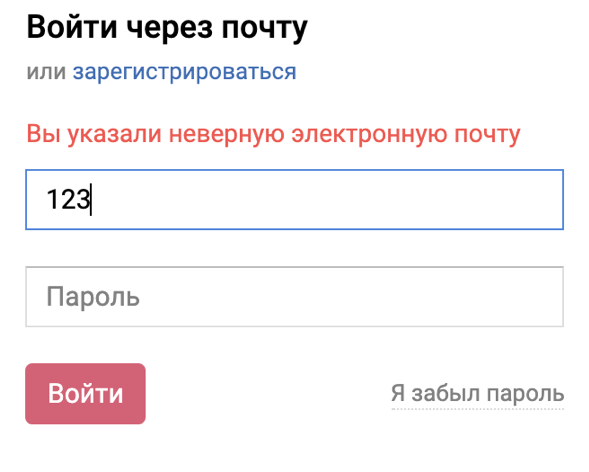
вместо <input type="button"> — <button>,

вместо <div class="navigation"> — <nav> и так далее.

Они однозначно описывают свой смысл или назначение как для браузеров, так и для веб-разработчиков.

**Валидация**

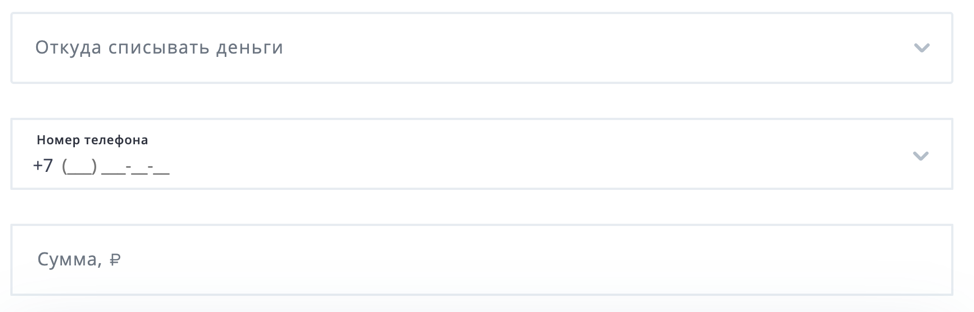
Наверняка, вы не раз встречались с тем, что когда хотели заполнить какие-то данные в форме, совершали ошибку и получали сообщение вроде «Поле заполнено неверно».



Проверка данных, введенных в форму, называется **валидацией**.

Существует **три** вида валидаций: мгновенная, по потере фокуса и по отправке формы.

1. **Мгновенная валидация** возможна только в тех случаях, когда в процессе ввода понятно, что значение некорректное. Обычно такие ошибки связаны с неправильной раскладкой клавиатуры (кириллица вместо латиницы) или вводом букв в цифровое поле (ip-адрес и тд.) Для этих случаев используются поля с масками (см. номер телефона);



2. **По потере фокуса** — когда пользователь убирает фокус с поля, которое заполняет (кликает по другому полю, например);

3. **По отправке формы** — при клике на кнопку submit (или любое другое действие, отправляющее форму).

Существует два разных способа проверки формы:

* **Проверка на стороне клиента** — при заполнении в браузере, прежде чем данные будут отправлены на сервер. Можно валидировать форму с помощью JavaScript либо использовать встроенную проверку — есть специальные функции проверки формы HTML5.
* **Проверка на стороне сервера** — возникает на сервере после отправки данных. Серверный код используется для проверки данных перед их сохранением в базе данных. Если данные не проходят проверку валидности, ответ отправляется обратно клиенту, чтобы сообщить пользователю, какие исправления должны быть сделаны.

Стремитесь к использованию комбинированной проверки на стороне клиента и сервера.

Способов валидации огромное множество. Например,  можно использовать:

* [css классы](https://developer.mozilla.org/ru/docs/Web/CSS/:valid) (и [здесь](https://developer.mozilla.org/ru/docs/Web/CSS/:invalid));
* атрибут [required](https://developer.mozilla.org/ru/docs/Web/HTML/Element/input#attr-required);
* регулярные выражения\* (атрибут [pattern](https://developer.mozilla.org/ru/docs/Web/HTML/Element/input#attr-pattern)).

И другие.

\***Регулярное выражение** (regexp) — это шаблон, который используется для соответствия комбинаций символов в текстовых строках, поэтому он идеально подходит для проверки формы (а также для многих других целей в JavaScript). Регулярные выражения довольно сложны.

Почитать больше о валидации [здесь](https://guides.kontur.ru/principles/validation/) и [здесь](https://developer.mozilla.org/ru/docs/Learn/HTML/Forms/%D0%92%D0%B0%D0%BB%D0%B8%D0%B4%D0%B0%D1%86%D0%B8%D1%8F_%D1%84%D0%BE%D1%80%D0%BC%D1%8B#:~:text=%D0%AD%D1%82%D0%BE%20%D0%BD%D0%B0%D0%B7%D1%8B%D0%B2%D0%B0%D0%B5%D1%82%D1%81%D1%8F%20%D0%B2%D0%B0%D0%BB%D0%B8%D0%B4%D0%B0%D1%86%D0%B8%D1%8F%20%D1%84%D0%BE%D1%80%D0%BC%D1%8B%20%E2%80%94%20%D0%BA%D0%BE%D0%B3%D0%B4%D0%B0,%D0%BF%D1%80%D0%B8%D0%BB%D0%BE).

ПРАКТИКА

Реализуйте слайдер для фото, исходя из требований ниже.

**Общие требования:**

* Должна быть возможность пролистывать фото. Например, добавить 2 кнопки вперед и назад (предыдущее, следующее и так далее);
* При клике «назад» на первом фото должно открываться последнее. При клике «вперед» на последнем фото должно открываться первое;
* Блок, в котором располагаются изображения, не должен меняться под размер картинок;
* Добавить анимацию при переключении фото;
* На входе — массив изображений. Их может быть любое количество. Нельзя завязываться на порядковый номер элемента;
* Картинки можно положить в папку asset. Либо использовать стороннее API, сделав GET запрос для получения изображений;
* Соблюдать семантическую верстку;
* Использовать селекторы по тегу для задания стилей нельзя. Использовать классы. В случае, когда есть необходимость — селектор по id;
* Дизайн произвольный. Минимум — застилизовать кнопки (добавить *hover*) и обеспечить для них *accessibility*: переключение по *tab*, *outline*, *cursor*.

При возникновении проблем по ходу решения вы всегда можете обратиться к ментору в *Slack*.

Удачи!

Это просто, чтоб не забыть

<button onclick="clickMe()">press1</button>

<button id="btn">press2</button>

<script>

function clickMe() {

alert('Clicked')

}

btn.onclick = clickMe;

</script>

</body>

</html>

AJAX, JSON, XML, XHR, PROMISEC, ASYNC/AWAIT, FETCH

**AJAX**(англ. Asynchronous JavaScript and XML — асинхронный JavaScript и XML) — это не технология сама по себе, а термин, который описывает «новый» подход к использованию существующих технологий вместе. АJAX включает: HTML или XHTML, CSS, JavaScript, DOM, XML, XSLT, и объект XMLHttpRequest.

Когда эти технологии объединяются в модель AJAX, web-приложения способны делать быстрые дополняющие обновления интерфейса пользователя без необходимости полной перезагрузки страницы браузером. Приложения работают быстрее и становятся более отзывчивыми к действиям пользователей.

## ****Форматы передачи данных****

Форматов передачи данных великое множество, и все они используются для определённых целей. Выше мы разобрали передачу HTML, CSS и JS.

Также существуют такие форматы данных, как **XML** и **JSON** — текстовые форматы, предназначенные для структурирования, хранения и передачи информации. Они выполняют одну и ту же роль, разница заключается в их представлении и обработке, которые мы далее рассмотрим.

## ****XML. Что это такое и как он устроен****

**XML**(с англ. eXtensible Markup Language — «расширяемый язык разметки») — текстовый формат, предназначенный для структурирования, хранения и передачи информации. Поддерживает хранение сложных структур данных. Язык называется **расширяемым**, поскольку он не фиксирует разметку, используемую в документах: каждый разработчик может создать разметку в соответствии с потребностями конкретной области.

XML похож на HTML, но у них разные предназначения. HTML решает проблему отображения данных, тогда как XML решает проблему структурирования и транспортировки данных. В HTML теги соответствуют документации HTML. В XML **вы сами придумываете теги**.

XML состоит из тегов, которые могут иметь атрибуты и включать в себя текст и другие теги.

**Пример XML**:

<book category="CHILDREN">

<title lang="en">Harry Potter</title>

<author>J. K. Rowling</author>

<year>2005</year>

<price>29.99</price>

</book>

## ****API для работы с XML****

Для работы с XML в JavaScript имеется класс [DOMParser](https://developer.mozilla.org/ru/docs/Web/API/DOMParser), который предоставляет экземпляр для работы с XML и HTML. Экземпляр предоставляет метод.

[DOMParser](https://developer.mozilla.org/ru/docs/Web/API/DOMParser) — это класс с интерфейсом парсинга HTML и XML.

parser.parseFromString() — преобразует XML в объект JavaScript в виде DOM.

Дальнейшая работа с полученным DOM представлением идёт с помощью методов поиска элементов в DOM и получения атрибутов элементов.

## ****JSON. Что это такое и как он устроен****

**JSON**(JavaScript Object Notation — Объектная нотация JavaScript) — текстовый формат обмена данными. JSON является синтаксисом для представления сложных структур, массивов данных различных типов в виде **строк**. По сравнению с XML занимает меньше объёма на диске за счёт отсутствия тегов разметки.

Описание JSON задокументировано в [стандарте RFC 4627](https://tools.ietf.org/html/rfc4627). Первоначально он был создан для JavaScript, но многие другие языки также имеют библиотеки, которые могут работать с ним. Таким образом, JSON легко использовать для обмена данными, когда клиент использует JavaScript, а сервер написан на Ruby/PHP/Java или любом другом языке.

JSON очень похож на объекты в JavaScript, но это не одно и тоже. Объекты в JavaScript — это тип данных и сущность языка, в то время как JSON — это обычная строка с определённым порядком символов, которую удобно преобразовать и из которой удобно получать данные. Объекты JavaScript можно преобразовать в JSON и наоборот, если JSON валидный. Невалидным JSON становится, если содержит недопустимые символы или лишние знаки.

**Пример JSON:**

{

"book": {

"category": "CHILDREN",

"title": {

"lang": "en",

"value": "Harry Potter"

},

"author": "J. K. Rowling",

"year": "2005",

"price": **29.99**

}

}

## ****API для работы с JSON****

Для работы с JSON в JavaScript имеется объект JSON (и в браузерах, и в Node.js), который предоставляет несколько методов для преобразования JSON в объекты JS и наоборот:

[JSON.stringify()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify) — преобразует объект JavaScript в строку JSON.

[JSON.parse()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse) — преобразует JSON в объект JavaScript.

## ****Введение в асинхронность. Как работает асинхронный код****

Асинхронное программирование в JavaScript — это работа с [event loop и очередью вызовов](https://developer.mozilla.org/ru/docs/Web/JavaScript/EventLoop). Это одна из концепций, которая делает JavaScript таким мощным и универсальным. Для продвижения дальше вам необходимо чётко понимать, как работает асинхронный код в JavaScript.

JavaScript  является однопоточным, но с возможностью реализации асинхронности.

**Однопоточный** — это значит, что JavaScript не может выполнять две функции одновременно. Все функции выполняются друг за другом в соответствии с их положением в очереди выполнения.

Другими словами, у нас есть очередь (очередь вызовов), в которую попадают все функции, выполняемые вашим кодом. И пока не выполнится предыдущая функция, следующая не начнёт свое выполнение.

В *JavaScript* есть два вида кода:

* 1. Синхронный,
  2. Асинхронный.

**Синхронный код**

Пример 1: все функции здесь синхронные и выполняются в порядке, согласно которому записаны в коде.

**function** f1() { console.log("1") };

**function** f2() { console.log("2") };

f2(); f1(); f2(); // В консоль выведется 2 1 2

**Асинхронный код**

Самый простой и давно существующий способ — асинхронные *callback*-функции.

Пример 2: здесь мы работаем с очередью вызовов. Стоит заметить, что, как и в первом примере, код в самих функциях f1 и f2 является синхронным.

**function** f1() { console.log("1") };

**function** f2() { console.log("2") };

**const** btn = document.querySelector("button"); // кнопка на странице

f2();

btn.addEventListener("click", f1);

f2(); // В консоль выведется 2 2.

// Когда мы совершим клик по кнопке, то функция f1 попадет в очередь

// вызовов и выполнится. В результате в консоль выведется 1

Это тот случай, когда событие пользователя (клик на кнопку) добавляет в очередь новую функцию для выполнения.

Асинхронный метод позволяет ставить в очередь выполнение первой функции, на пример f1(); f2(); f3(), я хочу выполнить f1(); после выполнения f2() и f3(), я вешаю на нее обработчик btn.addEventListener("click",f1); и только после нажатия на кнопку выполнится f1();, а до нее выполнятся f2() и f3().

Ссылка со скилфэкт видео по работе с файлами COOKIE

https://lms.skillfactory.ru/courses/course-v1:Skillfactory+FR+2020/courseware/f53e6b2f965d42b5850f008b8ec188aa/6b7bee5da6c9448f9c24efe127c3028c/6?activate\_block\_id=block-v1%3ASkillfactory%2BFR%2B2020%2Btype%40vertical%2Bblock%4031bb0b08034841749eeea417bab03d60

МОДУЛЬ 10 ЗАДАНИЕ 3

Эта итоговая практика потребует от вас комбинации алгоритмов решения как актуального модуля, так и ранее изученных тем (инструменты работы в браузере).

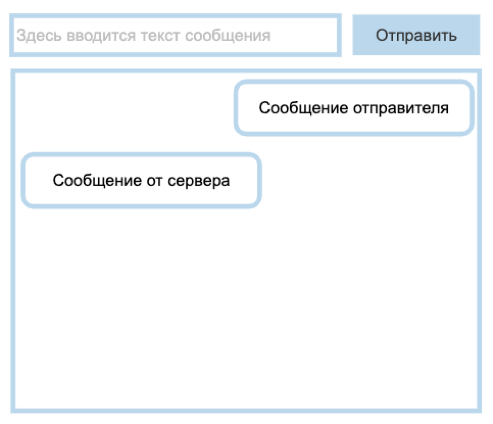
**Задание 3.**

* 1. Реализовать чат на основе эхо-сервера wss://echo-ws-service.herokuapp.com.

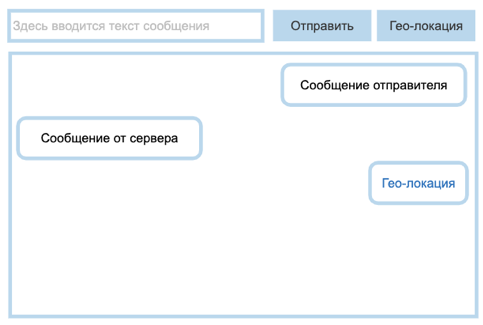
Интерфейс состоит из input, куда вводится текст сообщения, и кнопки «Отправить».

При клике на кнопку «Отправить» сообщение должно появляться в окне переписки.

Эхо-сервер будет отвечать вам тем же сообщением, его также необходимо выводить в чат:



* 1. Добавить в чат механизм отправки гео-локации:



При клике на кнопку «Гео-локация» необходимо отправить данные серверу и в чат вывести ссылку на <https://www.openstreetmap.org/> с вашей гео-локацией. **Сообщение, которое отправит обратно эхо-сервер, не выводить.**

Удачи!

/\*const wsUri = "wss://echo-ws-service.herokuapp.com";

const messageInput = document.querySelector('.message');

const sendBtn = document.querySelector('.j-btn-message');

const textWindow = document.querySelector('.text-window');

const geoBtn = document.querySelector('.j-btn-geo');

const placeholder = 'Здесь вводится текст сообщения';

// создаем объект вебсокет и описываем его поведение

let websocket = new WebSocket(wsUri);

websocket.onopen = function(evt) {

console.log("CONNECTED");

};

websocket.onerror = function(evt) {

console.log(evt.data)

};

websocket.onmessage = function(evt) {

console.log(evt.data);

addMessage(evt.data, 'flex-start');

};

//добавляем обработчик события при нажатии кнопки отправки сообщения

sendBtn.addEventListener('click', () => {

let message = messageInput.value;

websocket.send(message);

addMessage(message);

messageInput.value = ''

})

//функция для добавления сообщений в чат

function addMessage(message, position='flex-end') {

let element = `

<p class='message-window' style='align-self: ${position}'>

${message}

</p>

`;

let chat = textWindow.innerHTML;

textWindow.innerHTML = chat + element;

}

//описываем поведение при определении гео-позиции

const error = () => {

let error = "Позиция не может быть определена"

addMessage(error);

}

const success = (position) => {

const latitude = position.coords.latitude;

const longitude = position.coords.longitude;

let link = `https://www.openstreetmap.org/#map=18/${latitude}/${longitude}`;

addLink(link)

}

//функция для вставки ссылки в чат

function addLink(link) {

let element = `

<a href='${link}'

target='\_blank'

style='text-decoration: none;'

>

Гео-позиция

</a>

`;

let chat = textWindow.innerHTML;

textWindow.innerHTML = chat + element;

};

//добавляем обработчик события при нажатии кнопки гео-позиции

geoBtn.addEventListener('click', () => {

if (!navigator.geolocation) {

console.log("You can't use geolocation")

} else {

navigator.geolocation.getCurrentPosition(success, error);

};

});\*/

Пишет консоль.

Failed to load resource: net::ERR\_NAME\_NOT\_RESOLVED

main.js:29 CONNECTED

main.js:51 WebSocket is already in CLOSING or CLOSED state.

Network location provider at 'https://www.googleapis.com/' : Returned error code 403.