## JAVA SCRIPT

Оглавление

[JAVA SCRIPT 1](#_Toc530407467)

[Теория 3](#_Toc530407468)

[Область применения Java Script: 3](#_Toc530407469)

[Что умеет Java Script: 3](#_Toc530407470)

[Ограничения работы Java Script (что не умеет делать): 4](#_Toc530407471)

[История создания: 4](#_Toc530407472)

[Javascript сегодня: 5](#_Toc530407473)

[Место использования Javascript: 5](#_Toc530407474)

[Практика: 5](#_Toc530407475)

[Подключение Javascript 5](#_Toc530407476)

[Важные моменты теории: 6](#_Toc530407477)

[**alert** 7](#_Toc530407478)

[Типы данных: 9](#_Toc530407479)

[Массивы 11](#_Toc530407480)

[Математические операторы: 17](#_Toc530407481)

[Взаимодействие с пользователем сайта: 18](#_Toc530407482)

[Условия: 21](#_Toc530407483)

[Циклы: 23](#_Toc530407484)

[Прерывание циклов 24](#_Toc530407485)

[break 24](#_Toc530407486)

[continue 25](#_Toc530407487)

[Работа с массивами используя циклы 26](#_Toc530407488)

[Работа с объектами 26](#_Toc530407489)

[Конструкция Switch 27](#_Toc530407490)

[Функции 29](#_Toc530407491)

[Способы вызова функции: 29](#_Toc530407492)

[Порядок выполнения функции 31](#_Toc530407493)

[Область видимости переменных (Scope) 32](#_Toc530407494)

[Способы объявления события 42](#_Toc530407495)

[Наиболее употребляемые события: 43](#_Toc530407496)

[Поиск ошибок: 44](#_Toc530407497)

# Теория

Язык, который находиться на стороне клиента, а не сервера – как ПХП. Тоесть, он делает страницу динамической без запросов на сервер, что ускоряет процесс работы страницы.

Он делает то, что не может делать html - делает кнопки работающими, но при этом – умеет это делать без запросов на сервер с браузера, как это есть в случае с php.

**HTML –** говорит браузеру, как будет выглядеть страница. Тоесть, сервер шлет html-страницу, а браузер ее отображает.

**Java Script** – говорит браузеру, как эта страница будет себя вести.

**Java Script** интегрирован в браузер, поэтому знает всю информацию о браузере, которым пользуется клиент сайта.

**Java Script** подкючается в браузере напрямую с **html-файла** и выполняется напрямую без каких-либо запросов к серверу.

Область применения Java Script:

На нем пишут:

- веб-приложения

- браузерные операционные системы

- серверные приложения

- мобильные приложения

- виджеты (Веб-виджет (web widget))

*- это кусок кода, который может быть встроен пользователем в любую HTML страницу и использоваться без значительной модификации. Как правило, при создании веб-виджетов используются технологии DHTML, JavaScript и Adobe Flash*

Например, на страницах сайтов и блогов есть блоки, показывающие погоду, курсы валют, видеоролики с YouTube)

- прикладное программное обеспечение

- офисные программы (Excel)

Что умеет Java Script:

- изменять страницу

- добавлять или удалять теги

- изменять стили элементов

- применять стили элементам

- добавлять эффекты, анимации

- реагировать на события: Java Script может ожидать выполнения какого-нибудь события в браузере (клик мышкой), а потом среагировать выполнением функции

- проверять данные, которые вводит пользователь. Тоесть, авторизовать пользователя на стороне клиента, чтобы уменьшить нагрузку на сервер

- обращаться к серверу, чтобы подгрузить данные без перезагрузки страницы в браузере

- работать с локальным хранилищем данных, авторизовать данные и выводить сообщение

Ограничения работы Java Script (что не умеет делать):

- закрывать окна и закладки, которые были открыты не с его помощью

- не может защитить свой соарс-код: код, текст, картинки. Тоесть из страницы все можно легко скопировать

- не может осуществлять кросс-доменные запросы (в случае, когда нужные данные браузер должен подгружать с другого домена - это и есть кросс-доменные ajax-запросы.) без специального разрешения

- нет доступа к файлам, которые размещены на компьютере пользователя и доступа за рамки своей веб-страницы. Исключением являются лишь **файлы кукис**, которые **Java Script** сам записывает и считывает.

История создания:

В компании **Netscape** молодому инженеру Brendan Eich было дано задание придумать язык, который был бы похож на Java, но только, чтобы он был более легким к восприятию не программистов. Первый язык был придуман уже через 10 дней и назывался **Mocha**, потом его переименовали в **Live Script**. Не смотря на многие пробелы и погрешности этот язык выполнил главную задачу – вносить изменения в  **html-странице без ее перезагрузки.**

**В 1995 году** большинство компаний согласились использовать язык **Java Script** в своих проектах, как объектно-скриптовый язык с открытым стандартом.

Язык был стандартизирован ассоциацией **ECMA** и стал еще называться - **стандарт ECMAScript 5**

Javascript сегодня:

- действует новая редакция ECMAScript 6

- задействован с HTML5

- используется для написания десктопных приложений (Десктопные приложения – это программы, логика работы которых требует наличия оператора (человека работающего с программой), содержащие в себе всю полную функциональность и способные работать отдельно на любой машине изолированно от других приложений. Microsoft Word, Excel, Блокнот, однопользовательские игры – всё это примеры десктопных приложений. Для их работы необходимы лишь достаточные аппаратные ресурсы компьютера, само приложение и набор библиотек, содержащих функции для работы с приложением.)

- мобильных приложений

- серверной стороны

Место использования Javascript:

**1. При Front-end разработке:** HTML+CSS+Javascipt

**2.** **При Back-end разработке:** используя платформу Node.js

Практика:

Подключение Javascript

**1-й способ:** написать **тег <script>** в **html-файле** – используется лишь для написании небольших проектов и скриптов. Главным минусом является то, что он нагромождает **html-страницу** и делает ее менее читабельней. А при размещении его в секции <head> вообще останавливает загрузку **html-страницы**, пока не выполнится сам скрипт языка **Javascript**.

Сначала браузер будет читать все, что находиться в **теге <script>** до тех пор, пока не отработает **Javascript** код в нем**.**

<head>  
 <meta charset=**"UTF-8"**>  
 <title>**Javascript learning**</title>  
 <script type=**"text/javascript"**>  
 var *a* = "Hello!";  
 alert(*a*);  
 </script>  
</head>

**2-й способ:** создать внешний Javascript-файл и подключить.

1) В папке **domains /** в папке проекта (например, **mysite.localhost**)**/** создаем папку **script/**создаем файл с расширением **js.js**

2) Далее, напишем **js-код**, который был внутри тегов **<script>**, в этот файл. А в **html-файле** подключим этот файл с помощью вызова **тега** **<script** с арибутами **type=”text/javascript”** с атрибутом **src=”**адрес .js файла**/.js файл” ></script>** внутри секции **<body>**.

<body>  
<script type=**"text/javascript"** src=**"script/js.js"**></script>  
</body>

В связи с тем, что браузер останавливает загрузку всего **html-файла**, пока не будет выполнен **javascript-код**, **НАСТОЯТЕЛЬНО РЕКОМЕНДУЮТ** подключать **.js-файл** в конце **html-файла** в самом низу **секции <body>.**

Но также есть и **ИСКЛЮЧЕНИЯ**, когда js-файл нужно подключить в разделе (cекции) <head>. Например, когда нужно, чтобы Javascript-код отработал в самом начале до загрузки <html>-разметки.

<head>  
 <meta charset=**"UTF-8"**>  
 <title>**My First Page**</title>  
 <link rel=**"stylesheet"** type=**"text/css"** href=**"css/normalize1.css"**>  
 <link rel=**"stylesheet"** type=**"text/css"** href=**"css/style.css?v=1"**>  
 <script type=**"text/javascript"** src=**"script/js.js"**></script>  
</head>

Важные моменты теории:

1. Каждая строка кода **Javascript** – это инструкция для браузера.

2. Браузер читает код строка за строкой сверху вниз

3. Значок **;** - означает, что инструкция завершена и дальше начинается новая.

4. Все идентификаторы – регистрозависимые.

5. Переменная состоит из имени и куска памяти, который этой переменной будет задан.

**Чтобы объявить переменную нужно:**

- прописать ключевое слово:

**a) var**

**б) let (начали использовать с новой версии JS с 2015 )**

**в) const**

- прописать имя переменной var: **myMessage;**

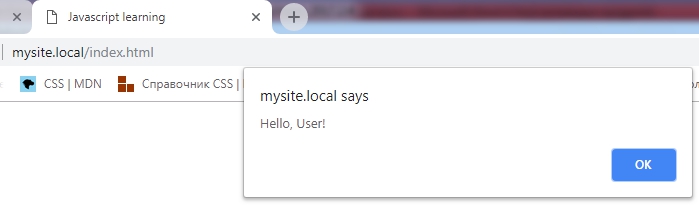
- после объявления переменной запишем в нее данные:

var *myMessage*;  
*myMessage* = "Hello, User!";

- обратимся к переменной по ее названию:

var *myMessage*;  
*myMessage* = "Hello, User!";  
alert (*myMessage*);

Выведет:



[**alert**](https://learn.javascript.ru/uibasic#alert)

Синтаксис:

alert(сообщение)

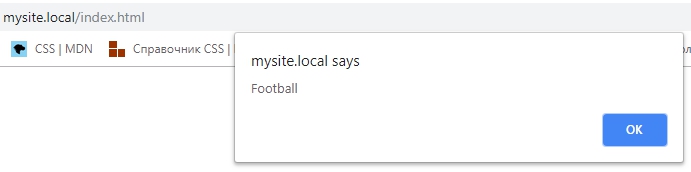
alert выводит на экран окно с сообщением и приостанавливает выполнение скрипта, пока пользователь не нажмёт «ОК».

alert( "Привет" );

Окно сообщения, которое выводится, является *модальным окном*. Слово «модальное» означает, что посетитель не может взаимодействовать со страницей, нажимать другие кнопки и т.п., пока не разберётся с окном. В данном случае – пока не нажмёт на «OK».

**6. Данные можно переписать:**

var *myMessage*;  
*myMessage* = "Hello, User!";  
alert (*myMessage*);  
  
*myMessage* = "Football";  
alert (*myMessage*);



**ПРИМЕЧЕНИЕ!**

В отличии от PHP значком **var** объявляем переменную, и дальше прописывать ее не нужно.

Пример в ПХП – мы не можем прописать переменную без знака $:

$i = 0;  
while($i < 3){  
 echo '<br>'.$i;  
 $i++;  
}

А в Javascript – можем:

var *i* = 0;  
while(*i* < 3){  
 alert(*i*);  
 *i*++;  
}

7. **Для имен существует 3 ограничения**:

1) имя может состоять из букв, цифр, символов $ и \_

2) названия не могут начинаться с цифр

3) нельзя использовать зарезервированные Javascript-ом слова

**8. Правила выбора имени переменных:**

- имена переменных должны максимально точно отображать суть данных, которые лежат в этих переменных

- никакого транслита – только английские слова, чтобы разработчики с любой точки мира могли прочитать ваш код

- короткие имена можно давать лишь тем переменным, которые не несут смысловой нагрузки: например, индексам

- переменные, которые состоят из нескольких словом нужно записывать способом: camelCase

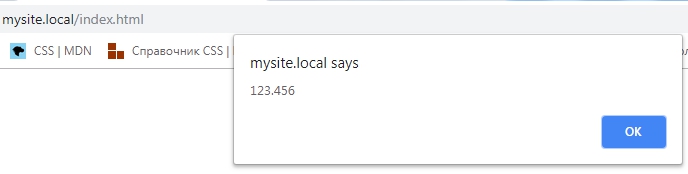
Типы данных:

**1. Число «number»:**

var *number* = 123;  
var *anotherNumber* = 123.456;  
alert (*anotherNumber*);

Этот тип используется как для целых, так и дробных чисел.

Выведет:

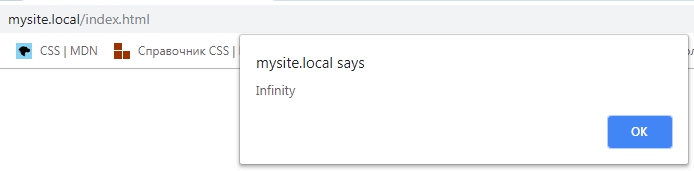


Еще существуют **специальные числовые значения:**

- infiniti (бесконечность):

alert (1/0);

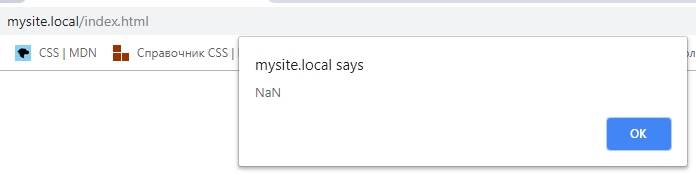
Когда делим на ноль, то выведет значение бесконечность в браузере.



- NaN (Not a Number):

alert (1\*"notANumber");

Когда умножаем на нечисловое значение, то выведет значение Not a Number в браузере.



**2. Строка «string»:**

Строки создаются при помощи двойных или одинарных кавычек:

var text = "моя строка";

var anotherText = 'еще строка';

var str = "012345";

В JavaScript нет разницы между двойными и одинарными кавычками.

**3. Логический тип (булев) Boolean:**

У него могу быть всего два значения: **true** или **false**

**Пишутся без кавычек:**

var *checked* = true;

**4. Специальное значение «null»:**

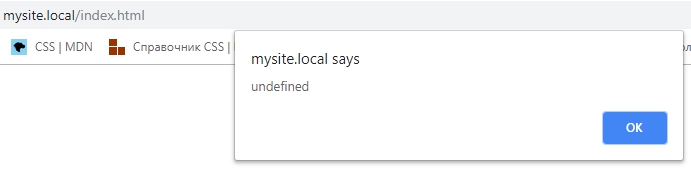
означает, что переменна была объявлена, но ее значение неизвестно.

var *age* = null;

**5. Специальное значение «undefined»:**

var *x =* undefined;  
alert(*x*);

Если переменная была объявлена, но ей не было присвоено никакого значения – означает, что переменная является **неопределенного типа**.

****

**Все эти типы данных называются примитивами.**

**Поэтому отдельно отмечается шестой тип данных:**

**6. Объекты «Objects»:**

**Структура:**

var user = {name: “Вася”}

имя объекта свойство значение

**Объекты** существуют для хранения данных и объявления более сложной сущности.

Например: в объекте мы можем хранить массив данных, а также другие объекты:

var *obj* = {  
 name: "John",  
 surname: "Smith",  
 jobs:{  
 first: "Makler",  
 second: "Broker"  
 }  
};

Массивы

Используются для того, чтобы хранить в одной переменной большое количество значений.

Записываются в **[ ] через запятую**

var *arr* = ['Bogdan', 'Corosa', 'MAZ'];

**Бывают смешанного типа, в котором могут существовать все типы данных :**

var *anotherArr* = [1, 'bus', true];

**Чтобы обратиться к массиву, тоесть, вывести его элемент на экран браузера необходимо:**

указать индекс элемента [0]

var *arr* = ['Bogdan', 'Corosa', 'MAZ'];  
alert(*arr*[0]);

**Чтобы переписать значение элемента необходимо:**

указать элемент массива и присвоить новое значение

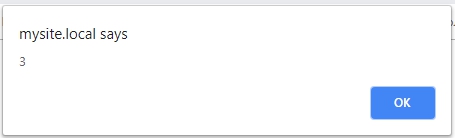
var *arr* = ['Bogdan', 'Corosa', 'MAZ'];  
*arr*[0] = 'LAZ';  
alert(*arr*[0]);

**Чтобы узнать длинну массива необходимо:**

использовать метод **length**

var *arr* = ['Bogdan', 'Corosa', 'MAZ'];  
alert(*arr*.length);

Выведет:

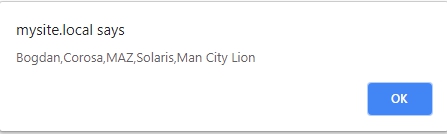


**Чтобы добавить новые элементы в массив:**

используем метод **.push()**

var *arrBuses* = ['Bogdan', 'Corosa', 'MAZ'];  
*arrBuses*.push('Solaris', 'Man');  
alert(*arrBuses*);

Выведет:

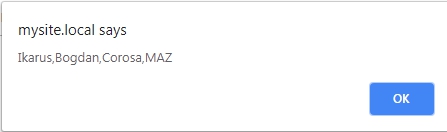


**Чтобы добавить новые элементы в начало массива:**

ипользуем метод **.unshift()**

var *arrBuses* = ['Bogdan', 'Corosa', 'MAZ'];  
*arrBuses*.unshift('Ikarus');  
alert(*arrBuses*);

Выведет:



**Чтобы удалить последний элемент из масива:**

используем метод **.pop()**

var *arrBuses* = ['Bogdan', 'Corosa', 'MAZ'];  
*arrBuses*.pop();  
alert(*arrBuses*);

Выведет уже без последнего элемента MAZ:



**Чтобы удалить первый элемент из масива:**

используем метод **.shift()**

var *arrBuses* = ['Ikarus','Bogdan', 'Corosa','MAZ'];  
*arrBuses*.shift();  
alert(*arrBuses*);

**Чтобы вывести содержания переменной в консоле браузера, необходимо:**

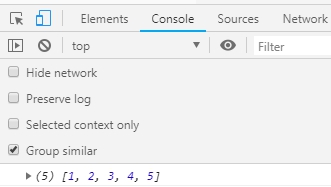
var *arr* = [1,2,3,4,5];  
console.log(*arr*);

- прописать метод console.log (название переменной)

- в браузере нажать F12

- зайти в окно Console

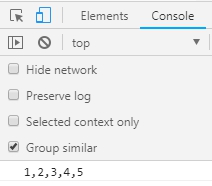
Выведет:



**Чтобы массив переделать в строку необходимо:**

<script>  
 var *arr* = [1,2,3,4,5];  
 var *str* = *arr*.join()*//создали новую переменую и в ней использовали метод join* console.log(*str*); *//вывели в консоле массив в виде строки*</script>

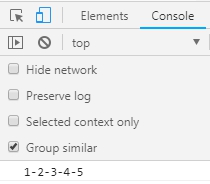
Выведет:



**Чтобы массив передать в строку и его элементы еще и отделить друг от друга чем-то инным, кроме запятой, то можно:**

<script>  
 var *arr* = [1,2,3,4,5];  
 var *str* = *arr*.join('-');*//добавили к методу join(любой символ в виде строки);* console.log(*str*);*//*</script>

Вывело в консоле:

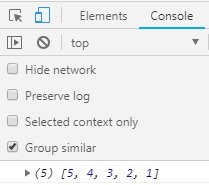


А вообще, по-умолчанию, выводить элементы строки будет через запятую.

**Чтобы массив переделать в строку и вывести ее в обратном порядке необходимо:**

<script>  
 var *arr* = [1,2,3,4,5];  
 *arr*.reverse();  
 console.log(*arr*);  
</script>

Выведет:



**Чтобы посортировать элементы массива в виде чисел, необходимо использовать функцию:**

**arr.sort(function: (a,b)**

**return a-b;**

**});**

которая возращает разницу между соседними элементами массива. И если разница будет иметь позитивное значение – тогда элементы меняются местами, а если отрицателное или ноль – остаются на месте.

<script>  
 var *arr* = [15,6,1,2,3,4,5];  
 *arr*.sort(function(a,b){  
 return a - b;  
 });  
 console.log(*arr*);  
</script>

Выведет в консоле:



**Чтобы посортировать элементы массива в виде строки, необходимо использовать функцию:**

**arr.sort();**

которая посортирует значение в алфавитном порядке.

<script>  
 var *buses* = ["LAZ", "MAZ", "Solaris", "Bogdan", "Corosa"];  
 *buses*.sort();  
 console.log(*buses*);  
</script>

Выведет:



**Чтобы вернуть диапазон элементов из массива (или строки) необходимо использовать функцию:**

**array.slice(beginSlice, endSlice);**

Метод slice возвращает часть строки от beginSlice до endSlice, но не включая символ под номером endSlice, не меняя вызывающую строку.

<script>  
 var *buses* = ["LAZ", "MAZ", "Solaris", "Bogdan", "Corosa"];  
 var *cityBuses* = *buses*.slice(2,4);  
 console.log(*cityBuses*);  
</script>

Выведет диапазон массива с индексами номер [2] до номера [4] не включая последний :



**Чтобы вставить в массив одни элементы и удалить другие одновременно, необходимо использовать метод:**

**array.splice(указать место вставляемых элементов,**

**указать количество элементов для удаления после вставки,**

**указать добавляемые элементы);**

<script>  
 var *buses* = ["LAZ", "MAZ", "Solaris", "Bogdan", "Corosa"];  
 *buses*.splice(0,4,"Ikarus","Man");//вставить в индекс под номером[0]  
 console.log(*buses*);  
</script>

Выведет:



Еще пример:  
var myFish = ['angel', 'clown', 'mandarin', 'sturgeon'];

var removed = myFish.splice(2, 0, 'drum');

// myFish равен ["angel", "clown", "drum", "mandarin", "sturgeon"]

**Чтобы удалить из массива несколько элементов, используем тот же метод:**

**array.splice(указываем место с которого начинаем удалять,**

**указываем количество элементов для удаления);**

<script>  
 var *buses* = ["LAZ", "MAZ", "Solaris", "Bogdan", "Corosa"];  
 *buses*.splice(1,1);  
 console.log(*buses*);  
</script>

Выведет:



var myFish2 = ['angel', 'clown', 'mandarin', 'sturgeon'];

var removed2 = myFish2.splice(2);//удаляем все, что идет после индекса номер [2] включительно

// myFish равен ["angel", "clown"]

Математические операторы:

**1. Стандартные операторы:** + , - , / , \*

var *a* = 3;  
var *b* = 4;  
var *sum* = *a* + *b*;  
console.log(*sum*);

Выведет в коносле : 7

Слаживать можно и строки:

var *a* = 'New ';  
var *b* = 'bus';  
var *sum* = *a* + *b*;  
console.log(*sum*);

Выведет в коносле: New bus

Тоесть с помощью знака **+** также происходит **конкатенация строк.**

Еще пример конкатенации строк, когда одна из них число, а вторая строка:

var *a* = 1;  
var *b* = '00';  
var *sum* = *a* + *b*;  
console.log(*sum*);

Выведет: 100

**2. Инкримент: ++**

var *a* = 10;  
*a*++;  
console.log(*a*);

Выведет в консоле: 11

**3. Декримент: --**

var *a* = 10;  
*a*--;  
console.log(*a*);

Выведет: 9

**4. Операторы сравнения: <, >, <=, >=, ==, ===**

- эти операторы не выполняют математичеких операций или операций с присвоения, а возвращают логическое значение – **true** или **false**

**=** означает **присваивание**

**==** означает **сравнение**

**===** означает **сравнение типов данных (число или строка)**

Например:

console.log('1' == 1);  
console.log('1' === 1);

Выведет в консоле:

true (потому, что: единица равна единице)

false (потому, что: строка не равна числу)

**5. Оператор “не равно”: !==** или **!=**

console.log( ‘1’ !=== 1);

**5. Оператор +** также используеться для:

- преобразовывание введенных пользователем данных в поле в числовое значение

var *year* = +prompt('When Javascript first appeared?', '');  
if(*year* === 1995){  
 alert('You are right!');  
}else{  
 alert('Wrong answer!');  
}

- конкатенации строк

var *bus* = prompt('Choice your bus: ', ‘ ’);*//можно еще указать подсказку, как пример: prompt('Choice your bus: ', 'Ikarus');*alert('Your bus is: ' + *bus*);

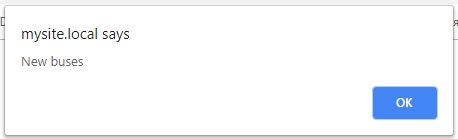
Взаимодействие с пользователем сайта:

Существуют базовые UI операции (“User Interface” – пользовательский интерфейс), которые позволяют работать с данными, полученными от пользователя:

**1. alert (“Your message is here”);** - выводит на экран модальное окно и блокирует работу с сайтом до тех пор, пока не выполниться действие с данным окном. Например, не нажмете кнопку **Оk**

alert('New buses');

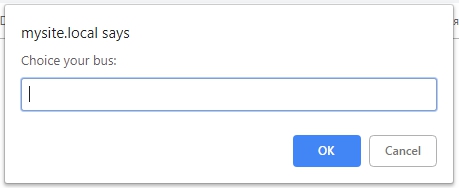
Выведет:



**2. prompt(title,default);** - выводит на экран модальное окно с полем, обязательным для заполнения. После нажатия **кнопки ОК** – выведет новое окно с результатом заполнения предидущего окна.

var *bus* = prompt('Choice your bus: ', ‘ ’);*//можно еще указать подсказку, как пример: prompt('Choice your bus: ', 'Ikarus');*alert('Your bus is: ' + *bus*);

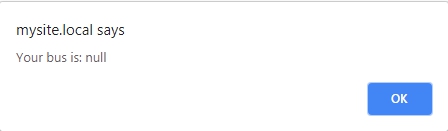
Выведет сначало:



А после заполнения поля и нажатия Ок:



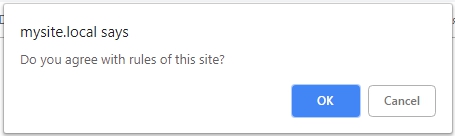
Но, если нажать **кнопку Cancel**, то выведет окно с информацией, что переменной было присвоенно значение **null**.



**3. confirm(question);** - выводит на экран модальное окно с вопросом.

var *agreeWithRules* = confirm('Do you agree with rules of this site?');  
alert(*agreeWithRules*);

Сначало выведет:



Если нажать **кнопку ОК**, то переменной будет присвоенно значение **true**, а если **– конпку Cancel**, то выведет окно с информацией, что переменной было присвоенно значение **null**.



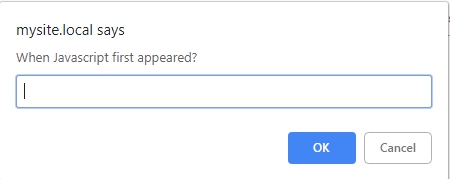
Условия:

в случае, когда код должен выполниться при выполнения условия – используется оператор **if,**

**if…else,** **elseif**

var *year* = +prompt('When Javascript first appeared?', '');  
if(*year* === 1995){  
 alert('You are right!');  
}else{  
 alert('Wrong answer!');  
}

Сначало выведет:



При правильном ответе – выполниться первое условие и на экран выведет первое модальное окно с фразой **‘You are right!’:**



**Сокращенный вариант записи конструкции условия:**

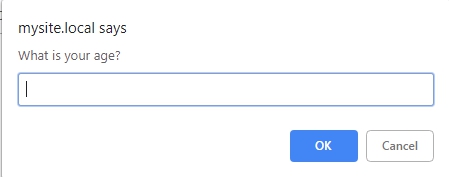
**1. Сначало обычный пример записи:**

var *age* = prompt ('How old are you?', '');*//создаем модальное окно без подсказки в '' и присваиваем переменной age*var *access*;*//создаем переменную access, которой позже присвоим значение*if(*age* > 17){  
 *access* = true;*//в случае выполнения условия - переменной access будет присвоенно значение true*}else{  
 *access* = false;*//иначе - false*}  
  
alert(*access*);*//создаем модальное окно, которое выведет значение переменной access*

**2. Теперь сокращенный пример записи:**

var *age* = prompt('What is your age?', '');  
var *access*;  
  
*access* = *age* > 17 ? true : false;*//вместо if используем ?, а вместо else - :  
//Если условие выполниться то переменной access присвоиться true, иначе - false*alert(*access*);

Выведет сначало вопрос:



А потом выполнение условия:



**Пример с лабораторной работы LVIV School IT**

Создать функцию, которая вернет булевый результат true – если число будет четное, и false –нечетное.

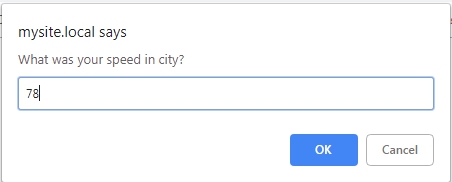
function isEven(number) {  
  
 if (number % 2 === 0) {  
 return true;  
 } else {  
 return false;  
 }  
  
}

var *getResult* = isEven(10);  
console.log(*getResult*);  
alert(*getResult*);

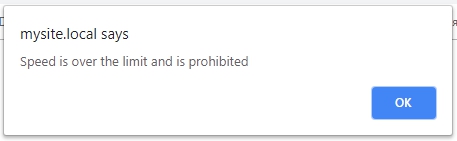
**Конструкция if ….else if,** в отличие от языка php записываеться **с пробелом** между **else** и **if**

let *speed* = +prompt('What was your speed in the city?', '');  
if (*speed* <= 50) {  
 alert('Speed is within the rules');  
} else if (*speed* > 50 && *speed* < 70) {  
 alert('Speed is over the limit, but is not rules violations!');  
} else {  
 alert('Speed is over the limit and is prohibited');  
}

**Выведет:**

****

А потом:

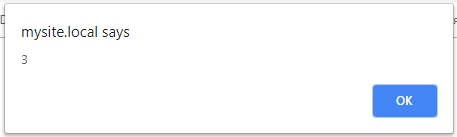


Циклы:

**1. while:**

let *i* = 1;  
while(*i* <= 3){  
 alert(*i*);  
 *i*++;  
}

Выведет окна со значением 1, потом 2, потом 3:



**2. do while** существует для того, чтобы на 100% быть уверенным, что тело цикла выполниться хотя бы один раз**:**

let *it* = 4;  
do {  
 alert(*it*);  
 *it*++;  
} while (*it* <= 3);

Все равно выведет 4, хотя условие потом и говрит, что it должно быть меньше или равно 3

**3. for**

for(it = 0; it <=3; it++){  
 alert(it);  
}

также можно записать другим вариантом:

let *it* = 1;  
for(; *it* <=3; *it*++){  
 alert(*it*);  
}

Прерывание циклов

Работу циклов можно останавливать не только с помощью не выполнения условий **if, if … else if, while, do…while**, но и с помощью **директивы break и continue.**

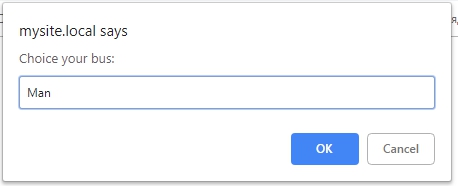
# break

Например:

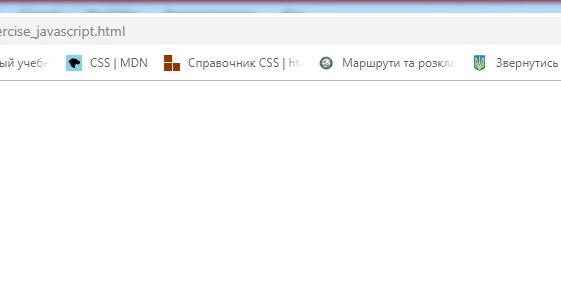
создадим цикл из которого нам нужно будет выйти, чтобы прервать выполнение кода, когда пользователь нажмет на кнопку **“Cancel”** в модальном окне.

while (true) {  
 let *bus* = prompt('Choice your bus: ', '');  
 if (*bus* == null) {*// если пользователь нажмет Cancel, то значению bus присвоиться значение null* break; *//и директива break завершит работу вечного цикла с условием true* }  
 alert('Your bus is: ' + *bus*);  
}

До тех пор, пока пользователь не нажмет кнопку “Cancel” – ему постоянно будет выводить модульное окно до бесконечности:



Но, как только он нажмет кнопку “Cancel”, то окно уже не появиться:



# continue

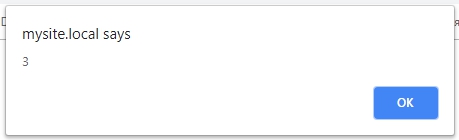
- служит для того, чтобы пропускать некоторые итерации, но при этом не выходя из цикла полностью, как в случае с директивой **break**

**Директива continue –** останавливает выполнение текущей итерации и сразу перебрасывает в следующую.

for (it = 1; it < 4; it++) {  
 if (it === 2) {*// если значение it будет равно два (итерация тоже будет идти второй* continue; *// тогда выполнение кода не будет выполняться, а директива continue перебросит сразу на третью итерацию* }  
 alert(it);  
}

Выведет сначала первое модульное окно со значением 1, 

но при нажатии кнопки “Ok” - второе не выведет, а сразу перейдет к третьему со значением 3 и так далее.



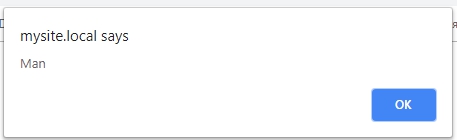
А вот, если бы в данном примере вместо директивы **continue** стояла бы **break**, то модульное окно прекратило бы выводиться сразу же после первого вывода модульного окна со значением 1.

Работа с массивами используя циклы

С помощью циклов можно перебирать массивы:

let *buses* = ["Man", "Maz", "Laz", "Solaris", "Bogdan"];  
for (i = 0; i < *buses*.length; i++) {*// задаем в условие, что итерация будет меньше значения длинны массива.Тоесть, мы будем выводить содержимое массива до тех пор, пока в них не закончаться элементы.* alert(*buses*[i]);  
}

Сначало выведет модульное окно с первым элементом:

******

А потом такие же окна поочереди со всеми элементами массива: Maz, Laz….Bogdan

Еще один пример, где нужно завести в пустой массив числа от 100 до ноля включительно:

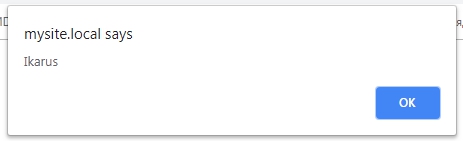
var *start* = 100;  
var *result* = [];  
while(*start* <=100 && *start* >=0){  
*result*.push(*start*);  
*start*--;  
}  
console.log(*result*);

Работа с объектами

Для того, чтобы перебрать массивы, которые находяться в объекте используется **итератор for (in)**, который работает так же , как в PHP – foreach:

let *buses* = {  
 name: "Ikarus",  
 model: "256",  
 year: 1989,  
 country: "Hungary"  
};  
  
for(let *key* in *buses*){  
 alert(*buses*[*key*]);  
}

Выведет все содержимое объекта по очереди:



….256, 1989, Hungary

Конструкция Switch

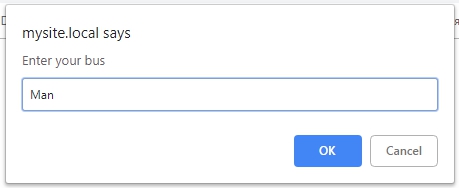
Данная конструкция заменяет конструкцию с **if…else** **if (с условием ==, а не => или <=)**

let *bus* = prompt("Enter your bus", "");  
if (*bus* == 'Man') {  
 alert("Germany");  
} else if (*bus* == 'Bogdan') {  
 alert("Germany")  
} else if (*bus* == 'Solaris') {  
 alert('Poland');  
 } else {  
 alert("Rest of world");  
 }

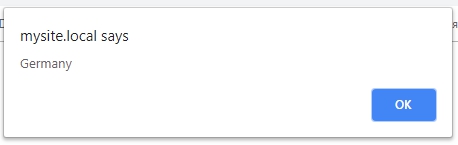
на **switch**, делая ее кароче:

let *bus* = prompt("Enter your bus","");  
switch(*bus*){  
 case 'Man':  
 alert('Germany');  
 break;  
 case 'Bogdan':  
 alert("Ukraine");  
 break;  
 case 'Solaris':  
 alert("Poland");  
 break;  
 default:  
 alert("Rest world");  
}

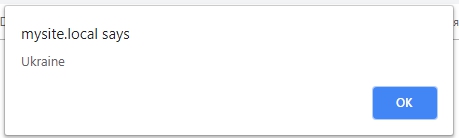
Директива break в данном коде необходима для того, чтобы код не повторялся дальше и не выводил модульные окна поочереди:



После нажатия Ок – сначало выведет:



А потом все остальные окна поочереди:



Poland и Rest of world.

**Пример с лабораторной работы LVIV School IT**

**Нужно создать калькулятор:**

function count(num1, num2, mark) {  
  
 var result;  
 switch (mark) {  
 case "+":  
 result = num1 + num2;  
 break;  
 case "-":  
 result = num1 - num2;  
 break;  
 case "\*":  
 result = num1 \* num2;  
 break;  
 case "/":  
 result = num1 / num2;  
 break;

default:  
 result = "Введите правильный параметр";

}  
 return result;  
}

**Выведем в консоль и на экран:**

ar *resultFunc* = count(110, 17, "-");  
console.log(*resultFunc*);  
alert(*resultFunc*);

Укороченная форма калькулятора:

function count(num1, num2, mark) {  
 switch (mark) {  
 case "+":  
 return num1 + num2;//break не нужен, так как return и так убивает код после  
 case "-": себя  
 return num1 - num2;  
 case "\*":  
 return num1 \* num2;  
 case "/":  
 return num1 / num2;  
 }  
 return 'Такого параметра не существует!';//если компилятор дошел аж до этого места, то значит что ввели неприемлемый параметр к калькулятору  
}

Функции

нужны для того, чтобы повторять те же самые действия много раз. Тоесть, функции – эти строительные блоки, которые содержат рабочий кусок кода. Главная задача функций – избавиться от дублирования кода.

Функции создаются один раз и могут вызываться много раз в любых местах.

Функция состоит из:

* слово **function** – служит для создания функции
* имени **count**(например) – служит для того, чтобы создать переменную, которой будет присвоен объект функции.

Название функции должно четко отображать ее задание, поэтому она должна иметь уникальное и понятное имя

* в круглых скобках **()** находятся параметры функции, которые будут локальными переменными функции
* в фигурных скобках **{}** находиться тело функции – блок кода, который будет выполнен после вызова функции

function count(num1,num2,num3){  
 let sum = num1+num2+num3;  
 console.log(sum);  
}

Способы вызова функции:

**1**. Способ **function declaration -**

**1.1.** Нужно написать имя функции и указать все ее входящие параметры (если они есть).

Так как мы можем задавать значение переменным и извнутри функции:

function count(){  
 let num1 = 1;  
 let num2 = 2;  
 let num3 = 3;  
  
 let sum = num1+num2+num3;  
 console.log(sum);  
}

Но сейчас расмотри пример, когды мы можем передать локальным переменным (тоесть тем, которые находяться и будут работать лишь внутри этой функции) значения с помощью входящих параметров функции:

function count(num1,num2,num3){  
  
 let sum = num1+num2+num3;  
 console.log(sum);  
}  
  
count(1,3,7);

В консоле выведет: **11**

В случае, **если мы не знаем точно придут ли нам входящие параметров или какое их будет точное количество (например, пользователь нам может передать имя, но без адреса),** тогда в средине функции оглашаем переменные таким образом:

function count(num1,num2,num3){  
 var num1  
var num1 = num1 || 1;  
var num2 = num2 || 2;  
var num3 = num3 || 3;//возьмет заданное нами запасное значение три  
 let sum = num1+num2+num3;  
 console.log(sum);  
 }  
  
 count(1,3);

Выведет: **7**

**ВАЖНЫЙ МОМЕНТ!!!!**   
Данный метод сработает лишь с использования **var**, так как **let** – будет считать, что параметр

num3 = false и запасной вариант просто не будет читать дальше, так как зачем это делать, если и так все ясно. А вот **var** будет читать код дальше и до конца.

**1.2.** Способ, когда функцию **можно вызвать** не только после ее создания, но и **до**:

count (1,3,7);

function count(num1,num2,num3){  
  
 let sum = num1+num2+num3;  
 console.log(sum);  
}

**2.** Способ **function expression** –

**2.1.** когда функцию мы присваиваем переменной, но при этом имя функции мы уже не пишем

let count = function (num1,num2,num3){  
 let sum = num1+num2+num3;  
 console.log(sum);  
}  
count(1,3,10);

Вызов осуществляем только после присвоения функции переменной.

Такой способ оглашения функций используеться, когда у функции есть зависимость от условий

**2.2.** Способ, **когда функция объявляется и сразу же вызывается –**

* **Без присвоения имя**

(function (){  
 let num1 = 1;  
 let num2 = 3;  
 let num3 = 7;  
 let sum = num1+num2+num3;  
 console.log(sum);  
})();

* **С присвоением имени**

нужна, когда функция вызывается извнутри. Это называется **рекурсивный вызов функции, когда функци вызывает сама себя.** Рекурсия используется для ситуаций, когда выполнение одной сложной задачи можно представить как некое действие в более простом варианте в совокупности с решением той же задачи.

Без примера пока что!

**3.** Способ, когда **в круглых скобках после параметров функции указывается и ее тело (в виде последнего параметра)** –

используется, когда нужно конструировать функцию во время выполнения программы, например, когда данные приходят с сервера или от пользователя.

let *sum* = new Function ('num1','num2','return num1+num2');  
console.log(*sum*(1,3));

**ВНИМАНИЕ! Параметры и само тело функции обязательно должны браться в кавычки ‘’.**

Порядок выполнения функции

Расмотри на примере:

function count(num1, num2){  
 let num3 = 7;  
 let sum = num1 + num2 + num3;  
 console.log(sum);  
}  
count(1,3);

1. После вызова функции **count()**, осуществляется этап ее инициализации.

**Инициализация** – это создание, активация, подготовка к работе, определение параметров, приведение программы или устройства в состояние готовности к использовании.

1. Интепретатор создает пустой **объект** и заполняет его, инициализированными переменными внутри функции **num3**  и **sum**.
2. Исполнение функции –

* переменным присваиваются значения:

**num3 = 7**

**sum = num1+num2+num3**

* исполняются команды:

**console.log(sum);**

4. После выполнения функции **объект** и **переменные удаляются и память очищается.**

**Вывод:**

* Нельзя перегружать функции.
* Одна функция должна содержать не более одного действия.
* Параметры, которые передаются функции становятся ее локальными переменными, тоесть существуют лишь в пределах данной функции.
* Новые локальные переменные внутри функций тоже можно создавать
* Значение функции возвращаются с помощью ключевого слова **return**. Эта команда должна идти в самом конце функции, так как она завершает выполнение функции и команды, стоящие после нее – не выполняются.

Область видимости переменных (Scope)

**Scope –** это набор переменных, объектов и функций, к которым есть доступ с конкретной части программы.

1. **Локальная область:**

Функция может содержать переменные, которые были объявленны внутри нее или переданные извне. Такие переменные называются **локальными переменными функции** и являются доступными самой функции и вложеным в нее функциям (если есть в наличии). такие переменные живут до тех пор, пока выполняется сама функция.

Докажем это на примере:

function count(num1, num2) {  
 let num3 = 7;  
 let sum = num1 + num2 + num3;  
}  
  
count(1, 3);  
alert(sum);

Выведет ошибку, так как переменная **sum** не существует за пределами функции **count.**

Все, что происходит внутри функции называеться – **Локальной областью видимости.**

**Внутри** **функций** могут спокойно использоватся **цикли**, а **переменные** для них могут объявляться через запятую:

function circle() {  
 let i, total;  
 for (i = 1; i < 3; i++) {  
 total = i \* 10;  
 }  
 //…some rest code  
}

1. **Глобальная область (Window):**

Функции могут обращаться и к глобальным переменным, которые объявляются в **глобальной области видимости,** в которых переменные живут до тех пор, пока выполняеться вся программа.

let *bus* = "LAZ";  
  
 function showBus(){  
 let transport = *bus* + " is Ukrainian product";  
 console.log(transport);  
}  
let *resultOfFunction* = showBus();

Выведет в консоле: **Laz is Ukrainian product**

Как работает функция в данной ситуации: сначало она ищет переменную **bus**  в локальной области видимости, а когда не находит – то переходит в **глобальную область видимости** и идет сверху вниз по коду. Когда находит переменную  **bus,** тогда функция берет ее значение и вставляет во внутрь функции.

Но, если будет оглашенна переменная с точно таким же именем **bus внутри функции,** тогда именно эта переменная будет приоритетной для функции.

let *bus* = "LAZ";  
  
 function showBus(){  
 let bus = "Solaris";  
 let transport = bus + " is Ukrainian product";  
 console.log(transport);  
}  
let *resultOfFunction* = showBus();

Выведет соответсвенно: **Solaris** is Ukrainian product

**Пример с лабораторной работы Lviv School IT**

var *senseOfLife* = 42;  
  
function showVariable(otherSenseOfLife) {  
  
 if (otherSenseOfLife !== undefined) {  
 return otherSenseOfLife;  
 } else {  
  
 return *senseOfLife*;  
 }  
}  
  
var *showResult* = showVariable();  
console.log(*showResult*);  
alert(*showResult*);

**ВАЖНЫЙ МОМЕНТ!!!** Если внутри функции **объявить** **переменную** без использования ключевого слова **let** или **var,** тогда она станет **глобальной.**

let *bus* = "LAZ";  
  
function showBus() {  
 bus = "Solaris";  
 let transport = bus + " is Ukrainian product";  
 console.log(transport);  
}  
  
showBus();  
console.log(*bus*);

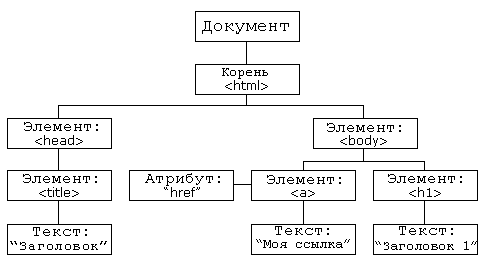
Выведет: два раза **Solaris,** но лишь при вызове функции **showBus**, так как именно вызов функции перезаписывает глобальную переменную **let bus = “LAZ”;** в **bus = “Solaris”**, которая была объявленна внутри функции **function showBus()**



DOM (Объектная модель документа)

После того как пользователь набрал доменный адрес в строке браузера и нажал Enter – сервер возвращает браузеру сначала html- страницу. После того, как она полностью загрузится в браузере – браузер создает DOM(Объектную Модель Документа) для этой страницы.

DOM - можно представить, как дерево элементов:



Браузер создает структуру страницы, сохраняя связи между элементами, и вынося в отдельные узлы – теги, атрибутты, текст, комментарии.

На схеме видно, что элемент <head> и <body> имеют разыне ветки, но при этом они сохраняют связсь друг с другом. Благодяря этому Javascript имеет доступ к каждому элементу страницы и полную власть над <html> документом и может:

* + менять html-элементы и атрибуты на странице
  + менять css стили
  + удалять html – элементы и артибуты
  + добавлять новые html – элементы и атрибуты
  + реагировать на события пользователя и создавать новые относительно html-документа

**Для того, чтобы модифицировать или удалить html-элементы:**

сначала к ним нужно обратиться и найти на html странице. **Обращаться можно за id, class, тегом.**

1. **Обратимся** **к** **элементу через id**, который имеет **id** со страницы **js:**

var *\_id* = document.getElementById("\_id");

где (“\_id”) – указываем имя идентификатора, которое мы задали в элементе <div> на html-странице

2. Теперь, когда мы присвоили **элемент** html страницы переменной в js-коде, мы можем его **поменять**. Например, поменяем его **css-стиль.**

**ВАЖНОЕ!!!**

Все стили, которые в **CSS**-языке писались через дефис, в **JavaScript’e** пишутся через **camelCase.**

*\_id*.style.fontSize = "20px";

Выведет увеличенный текст в блоке, к которому мы обратились через **Id**

3. **Обратимся** **к** **элементу** **через** **class:**

var *\_classArr* = document.getElementsByClassName("\_class");

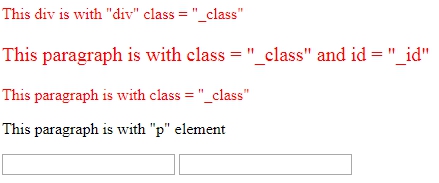
используя обращение по классу мы создадим массив из элементов, которые имеют точно такой же класс. И этот массив будет присвоен переменной **\_classArr**

4. Чтобы работать с этим массивом будем использовать цикл **for:**

for(var *i* = 0; *i* < *\_classArr*.length; *i*++ ){  
 *\_classArr*[*i*].style.color = 'red';  
}

с помощью которого мы проходимся по всем элементам массива и обращаемся к их стилям, меняя цвет на красный.

Выведет:



5. **Обратимся** **к элементу через тег:**

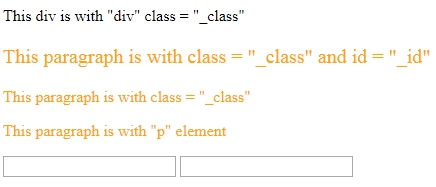
var *\_tagArr* = document.getElementsByTagName('p');

используя обращение к тегу <p> через команду **getElementsByTagName**, мы создадим массив со всеми параграфами в html-файле.

6. C помощью цикла for() будем обрабатывать массив, в котором находяться все элементы с одинаковым тегом <p>:

for(var *i* = 0; *i* < *\_tagArr*.length; *i*++){  
 *\_tagArr*[*i*].style.color = "orange";  
}

Выведет:



7. **Чтобы обратиться ко** всем элементам на html-странице, которые отвечают css-селектору (id, type, class, атрибут или значение **атрибута**) можно также:

использовать **метод querySelectorAll**

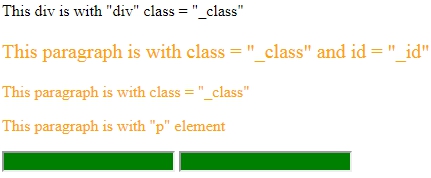
var *elements* = document.querySelectorAll('input');

В данном случае этот метод обращается к тегам input и из-за того, что их несколько - создаеться массив из них.

8. С помощью цикла for обратимся к стилю элментов массива и поменяем их цвет:

var *elements* = document.querySelectorAll('input');  
for(var *i*=0; *i* < *elements*.length; *i*++){  
 *elements*[*i*].style.backgroundColor = 'green';  
}

Выведет:



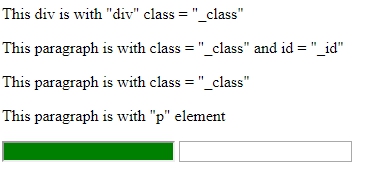
9. Чтобыобратиться **к первому по счету элементу внутри тега,** которыйотвечает css-селектору (id, type, class, атрибут или значение атрибута):

нужно использовать **метод querySelector**

var *elements* = document.querySelector("input");  
*elements*.style.backgroundColor = 'green';

И, так как мы обращаемся к одному элементу “input”, то мы не используем цыкл for, который нужен лищь при обращению к массиву элементов.

Выведет:



HTML DOM – методы(свойства)

- это действия, которые мы можем осуществлять с <html> элементами.

Все методы можно найти в интернете, но расмотрим основные:



**Пример 1.**

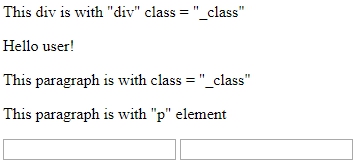
1. Получим доступ к параграфу с айдишником “\_id” и поменяем его текст:

var *\_id* = document.getElementById('\_id');

2. С помощью метода “innerHTML” – заменим текст параграфа на “Hello, user!”

*\_id*.innerHTML = "Hello user!";

Выведет:



**Пример 2.**

Мы можем не только менять <html> - элемент, но и **получать его значение.**

1. Например, в элемент <input> добавим атрибут **value:**

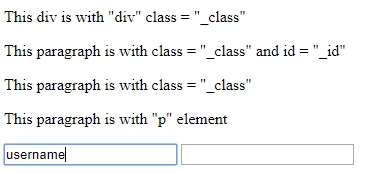
<input type = **"text"** value=**"username"**/>  
<input type = **"password"**/>

2. С помощью метода **querySelector** получим доступ к элементу <input>, а с помощью дописи к строке метода слова **.value** - также обратимся к его атрибуту **“value” со значением** :

let *value* = document.querySelector("input[type='text']").value;

Теперь переменной **value** будет присвоенно **значение** **атрибута** **value**, которое находиться в поле **input с типом «text».**

Выведет:



3. Также это значение мы можем присвоить другому тегу, например параграфу.

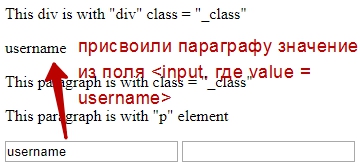
* Получим доступ к тегу с помощью метода **document.getElementById**

let *\_id* = document.getElementById("\_id");

* Присвоим значение **value** к тегу, доступ к которому мы только что получили за его **id**

*\_id*.innerHTML = *value*;

Выведет:



**Пример 3.**

В данном примере мы будем создавать элементы списка, добавлять их, удалять и менять одни элементы другими.

1. В html –файле создадим маркерованный список

<ul id = **'list'**>  
 <li>**First Item**</li>  
 <li>**Second Item**</li>  
 <li>**Third Item**</li>  
 <li>**Fourth Item**</li>  
</ul>

2. **Чтобы создать новый элемент:**

используем метод **document.createElement(); в котором укажем имя тега, который мы хотим создать**

В данном примере в js-файле создадим дочерний элемент к элементу **<ul>** и назовем его **child**

var *child* = document.createElement("li");

3. Но помимо тега, нам еще нужно создать в нем текст, для этого существует метод **document.createTextNode(‘’);**

var *text* = document.createTextNode("Fifth Item");

4. Полноценный элемент мы создали, а теперь добавим его на <html>-страницу. Для этого укажем, какому родительскому элементу он пренадлежит. В нашем примере – это элемент <ul>.

Поэтому сначало:

* **получим доступ** к родительскому элементу <ul> по его id

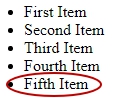
var *parent* = document.getElementById('list');

* **соеденим два созданных элемента**: “**li**” вместе с его текстом “**Fifth** **Item**” с помощью метода **.appendChild();** который сделает **элемент** с **текстом** дочерним элементом **тега** <**li**>
* var *child* = document.createElement("li");  
  var *text* = document.createTextNode("Fifth Item");  
  *child*.appendChild(*text*);
* Далее нужно элеменет **child** (<li >который уже содержит текст) **сделать дочерним** элементом **parent**

var *child* = document.createElement("li");  
var *text* = document.createTextNode("Fifth Item");  
*child*.appendChild(*text*);

var *parent* = document.getElementById('list');  
*parent*.appendChild(*child*);//<li>”Fifth element”</li> сделали дочерним <ul>

* Теперь данная конструкция добавит соедененный элемент <li>+его текст в конец списка дочерних элементов <ul>



5. В случае, если нам нужно **добавить** элемент не в конец списка, а, например, **вторым** **по** **счету** – для этого нам нужно собрать все уже существующие дочерние элементы списка <ul> внутри него, чтобы указать после какого именно элемента вставлять новый созданный элемент <li>. Для этого воспользуемся уже знакомым методом **document.getElementsByTagName();** и обратимся к всем элементом за названием тега

var *child* = document.createElement("li");  
var *text* = document.createTextNode("New Item");//поменяли название для нового тега  
*child*.appendChild(*text*);  
  
var *parent* = document.getElementById('list');  
var *liItems* = document.getElementsByTagName('li');

6. Теперь **добавим** новый элемент **child** в родительский єлемент **parent** с указанием конкретного места в нем. Для этого воспользуемся методом: **.insertBefore(newElement, before);**

где **NewElement** – имя нового елемента,

а **before** – имя существующего элемента списка , перед которым мы добавим новый элемент.

Новый элемент у нас присвоен переменной **child,** чтобы обратиться к элементу списка – нужно указать каким по счету он идет внутри массива **liItems** (мы его вызывали с помощью метода document.getElementsByTagname(‘li))

var *child* = document.createElement("li");  
var *text* = document.createTextNode("New Item");  
*child*.appendChild(*text*);  
  
var *parent* = document.getElementById('list');//вызвали список <ul>  
var *liItems* = document.getElementsByTagName('li');//вызвали массив с его элемен-ми <li>  
  
*parent*.insertBefore(*child*, *liItems*[1]);//вставили новый эл-нт перед вторым в списке

Выведет:

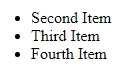


7. Далее мы **удалим** ненужные элементы:

Чтобы удалить элементы со списка, используем метод **.removeChild();**

var *parent* = document.getElementById('list');  
var *liItems* = document.getElementsByTagName('li');  
  
*parent*.removeChild(*liItems*[0]);//указали удалить 1 по счету элемент со списка “parent”

Выведет список без первого элемента:



8. Теперь **заменим элемент** один на другой:

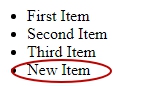
воспользуемся методом .**replaceChild(newElement, oldElement )**, где

**newElement** – имя нового элемента

**oldElement** – элемент, который будем заменять

var *child* = document.createElement('li');//создали новый элемент <li>  
var *text* = document.createTextNode('New Item');//создали текст  
*child*.appendChild(*text*);//прикрепили текст к элементу <li>  
  
var *parent* = document.getElementById('list');//получили доступ к списку <ul> по его id  
var *liItems* = document.getElementsByTagName('li');//получили доступ к его тегам  
  
*parent*.replaceChild(*child*,*liItems*[3]);//заменили 4 тег в списке <ul>

Выведет:



События

Как уже упоминалось ранее JavaScript создавался для того, чтобы сделать <html>-с траницу живой (динамичной). Поэтому JavaScript умеет реагировать на следующие события:

**1. Когда пользователь нажимает на конкретную кнопку или элемент**

**2. Загрузка страницы**

**3. Загрузка изображения**

**4. Пользователь провел курсором миши над элементом**

**5. Пользователь ввел данные в <input> поля или отправил форму**

**6. Пользователь набирает текст**

**7. Изменяется размер окна браузера**

**8. Многие другие события**

# Способы объявления события

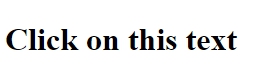
**1. Объявим событие, как атрибут элемента:**

Событие **onclick(нажать на кнопку курсором)** запишем виде атрибута тега **<h1>** и пропишем в нем команду **this.innerHTML**, которая будет менять его текст. Этот атрибут пишем в <html>- странице.

Ключевое слово **this** означает контекст (среда обитания) выполнения всей функции. Тоесть **this** обращается к элементу, по которому мы кликнем – **Click on this text!** и в контексте конкретно этого элемента должен будет выполниться код **Oops**!. А если не поставить **this,** то **onclick** выполниться для всех **<h1>**

<h1 id=**"title"** onclick=**"**this.innerHTML = 'Oops'**"**>**Click on this text!**</h1>

Увидим, что текст заменился с:



на:



**2. Присвоить атрибуту элемента, не всю команду this.innerHTML , а лишь функцию, которая будет выполнять эту же команду.**

* Тоесть, вместо команды **this.innerHTML**, мы пропишем функцию **changeText(this)**.

Когда будет исполнено событие **onclick**, тогда выполниться функция **changeText**, которая передаст ключевое слово **this** в контексте (в среде обитания), где должна выполниться функция. Тоесть в пределах именно этого элемента <**h1**>, исключая все остальные возможные <**h1**>

<h1 id=**"title"** onclick=**"**changeText(this)**"**>**Click on this text!**</h1>

* Далее в файле js.js мы создаем эту функцию, которая своим параметром (**element**) получает конкретный элемент (**this**). А потом с помощью команды **this** мы обращаемся ко всему тексту, который будет в атрибуте **Click on this text!**

function changeText(element){  
}

* В этой функции мы обращаемся к элементу и меняем текст на “Ooops!”:

function changeText(element){  
 element.innerHTML = "Oooops!";//меняем текст на Oooops!  
}

По сути – мы сначало:

* создали функцию, которая меняеет текст на Ooops! в файле JavaScript
* а потом вызвали ее внутри атрибута **Click** в файле HTML

Отработает так же само – сначало выведет **Click on this text!** а после нажатия – поменяет текст на **Ooops!**

**Другими словами this – это указатель на объект(window), в котором происходят все наши события – изменения текста в <h1> в данном примере.**

Еще раз пройдемся по тому, что мы написали:

- Когда выполнится событие **onclick,** тогда выполниться функция **changeText(),** которая передаст во внутрь себя входящий параметр **this – тоесть контекст, в котором должна выполниться функции.** В нашем примере контекстом являеться элемент **<h1></h1>**

- В функции мы прописали команду **innerHTML**, которая поменяет текст

Почему мы не можем использовать более простой метод **document.getElementById???**

Потому, что у нас может быть большое количество событий, которые вызывают одну и ту же функцию, поэтому лучше передавать контекст вызова (**this**). Тоесть тот конкретный элемент, где отработало событие, чтобы не потревожить остальные элементы.

**3. Объявление события без загромаждения html-файла лишними атрибутами, как onclick, а поиск элемента по id или другому селектору. И уже после этого менять ему текст, например.**

* в **html** файле создаем атрибут, над которым будем выполнено событие

<h1 id=**"title"**>**Click on this text!**</h1>

* в **js**-файле получим доступ к атрибуту с помощью обращения к селектору атрибута – **id**

document.getElementById("title")

* и в той же строке сразу же присвоим атрибуту событие .**onclick с функцией, которая будет выполнять определенные действия**

document.getElementById("title").onclick = function(){  
   
}

* функция будет изменять текст атрибута
* document.getElementById("title").onclick = function () {  
   this.innerHTML = "Ooops";  
  };

Выведет то же самое: сначало текст Click on this text, а после клика – Oooops!

### Наиболее употребляемые события:

* **onclick** – когда пользователь нажал курсором на элемент
* **onload** - когда загрузился документ или конкретная его часть
* **onchange** – для чекбоксов и радиобаттонов, когда меняеться их состояние
* **onmouseover, onmouseout** – когда пользователь проводит курсором над элементом
* **onmousedown, onmouseup** - когда пользователь нажал на кнопку, а потом отпустил клавишу мыши
* **onkeypress, onkeyup, onkeydown** – когда пользователь нажал на клавишу клавиатуры
* **onfocus, onblur** – когда пользователь вводит определенные данные в поля, а **onblur –** когда он сосредоточился на чем-то еще и текущий элемент уже не в фокусе.

### Пример событий:

Создадим заглавие, которое будет менять цвет при наведении курсора.

1. Создадим событие **onmouseout** внутри элемента <h1>Mouse over this text</h1>, благодаря которому заголовок будет менять свой цвет при наведении на него мышкой на красный, а когда уберем мышку – станет зеленым.

<h1 onmouseover = **"***style*.color = 'red'**"** onmouseout = **"***style*.color = 'green'**"** >**Mouse is over this text**</h1>

Выведет при наведении курсора на надпись:



А после того, как уберем курсор в сторону:



2. Создадим событие **onfocus**, благодаря которому поле <input> будет менять цвет своего фона при вводе в него данных. Для этого событию **onfocus** присвоим функцию, которая будет менять цвет фона. Самой функции нужно передать контекст ее исполнения. Как мы помним контекст передает команда **this**, которая говорит функции, что работать будем в предалах поля <input>

<label>**Enter your name:**</label>  
<input type=**"text"** onfocus=**"**myFunction(this)**"**>

Далее в **js-файле,** вызовем функцию, которая будет вызываться в контексте поля <input >. Для этого входящим параметром функции будет элемент, в контексте которого функция вызывается. А в теле функции пишем команду, которая будет менять цвет фона поля <input> на желтый.

function myFunction(element){  
 element.style.backgroundColor = 'yellow';  
}

Выведет желтое поля при заполнении его данными:

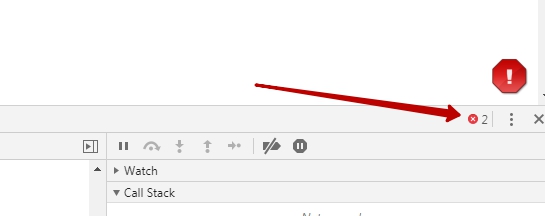


Поиск ошибок:

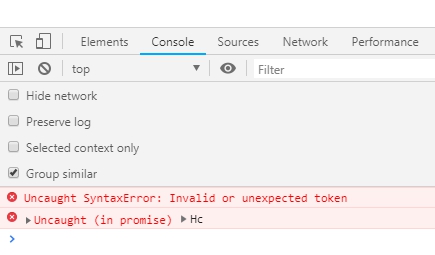
1. Вначале прописать **функцию debugger** в самом коде:

let 0map;//специально допустили ошибку для дебаггера  
 var *chicago* = {lat: 41.85, lng: -87.65};  
debugger;

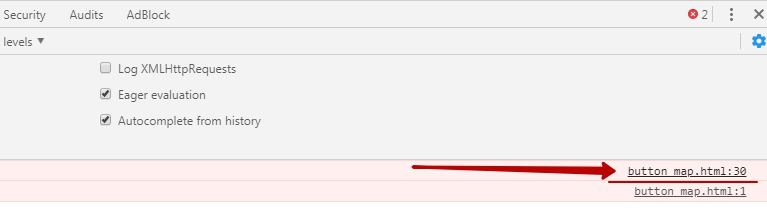
2. В браузере нажать F12 и увидим оповещение об ошибках в правом углу экрана:



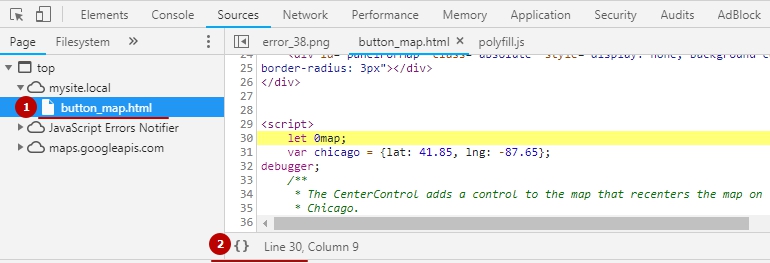
3. Заходим в окно Console и читаем сообщения об ошибках:



4. Открываем ссылки на ошибки справа в углу экрана браузера:



5. Увидим месторасположение файла с ошибкой, его название и номер ряда, в которой она допущена.

 6. Для сообщении об ошибке в коде JS мы установили плагин, котрый выведет красный знак в браузере:

