**ТИПЫ ДАННЫХ**

JavaScript язык с динамической типизацией. То есть тип данных переменной изменяется от значения, хранящегося в нем.

Есть 8 основных типов данных:

**Число (number)** – представляет как целочисленные значений, так и числа с плавающей точкой.

Имеются “специальные числовые значения”:

* **Infinity** – математическая бесконечность, больше любого числа. Его можно получить в результате деления на 0, либо задать явно.
* **-Infinity** – отрицательная бесконечность.
* **NaN (Not a number)** – результат неправильной или неопределенной математической операции. Любая математическая операция с NaN возвращает NaN. NaN не равен ничему, даже самому себе. Можно проверить с помощью функции **isNan(value)**.

Функция **isFinite(value)** возвращает true, если число не является NaN/Infinity/-Infinity.

Для явного преобразования можно использовать унарный плюс, либо Number(). Но они будут возвращать NaN если строка не является в точности числом.

Функции parseInt и parseFloat могут распарсить такое: “12.11dasdad” -> 12.11;

Такое не могут “dasd10” -> NaN.

**BigInt** – был добавлен, чтобы дать возможность работать с целыми числами произвольной длины. (number ограничен значением (253-1)).

Свойство Number.MAX\_VALUE представляет максимальное числовое значение, представимое в JavaScript. Значение больше представляются как бесконечность.

maxSaveInteger представляет максимальное безопасное целочисленное значение в JavaScript (2^53 - 1)

Чтобы число стало BigInt нужно добавить к литералу в конце символ “n”.

**Строка(String) –** должны заключаться в кавычки.

1. Двойные кавычки: "Привет".
2. Одинарные кавычки: 'Привет'.
3. Обратные кавычки: `Привет ${name}`. Позволяют встраивать результат выражения в строку.

**Boolean (логический тип)** – может принимать 2 значения: true либо false.

**null** – формирует отдельный тип. Специальное значение, которое представляет собой “ничего”.

**undefind** – значение не было присвоено. Можно присвоить undefind явно, но не рекомендуется. Обычно null используется для присвоения переменной «неизвестного» или «пустого» значения, а undefind – для проверок, была ли назначена переменная.

**object –** объект. Состоит из пар ключ-значение. Ключ (имя свойства) – строка, значение что угодно.

Имя свойства может состоять из нескольких слов через пробел, но тогда оно должно быть заключено в кавычки.

Пустой объект можно создать, используя один из двух вариантов синтаксиса:

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

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

Можем сразу создать объект из нескольких свойств:

let user = { // объект

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

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

};

Для обращения к свойствам используется запись через точку, либо через квадратные скобки и имя свойства в кавычках.

alert( user.name ); // John

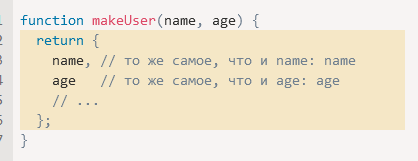
alert( user.age ); // 30

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

Для удаления свойства используется оператор **delete**

delete user.age;

Если хотим значение переменной установить как свойство объекта, и использовать то же имя что и переменная, можно использовать короткую запись



Имя свойства может содержать зарезервированные слова (for, let, return).

При обращении к свойству, которого нет в объекте возвращается underfind. Чтобы проверить, есть ли у объекта такое свойство можно использовать оператор **in**:

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

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

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

Мы можем менять объект, объявленый с *const*(добавлять, удалять поля и менять их значение). Мы не можем изменить только саму переменную.

Для перебора всех свойств объекта используется цикл **for..in**:

for (key in object) {

// тело цикла выполняется для каждого свойства объекта

}

Для проверки типа используется оператор **typeof**. Возвращает строку с именем типа.

**ОПЕРАТОРЫ**

Делятся на унарные и бинарные операторы.

**Математические**:

* Сложение +,
* Вычитание -,
* Умножение \*,
* Деление /,
* Взятие остатка от деления %,
* Возведение в степень \*\*.

Плюс может работать **по-разному**:

* При сложении двух строк происходит их конкатенация  
  let s = "моя" + "строка";

alert(s); // моястрока

* Если хотя бы один операнд является строкой, то второй тоже приводится к строке.  
  alert( 2 + '1' ); // "21"
* Унарный плюс преобразует значение в число
* Все остальные мат. Операции работают только с числами (возвращают NaN если не с числами).

**Оператор присваивания** “=” так же возвращает значение, благодаря чему его можно использовать в более сложных выражениях:

let a = 1;

let b = 2;

*let* c *=* *3* *-* *(*a *=* b *+* *1);*

alert( a ); // 3

alert( c ); // 0

**инкремент/декремент** “++/--” можно использовать только на переменной (5++ ошибка). Префиксная запись сначала увеличивает значение, а потом его возвращает. Постфиксная – сначала возвращает старое значение, а потом увеличивает. Если применить его не на числе вернется NaN

**Побитовые операторы**

* AND(и) ( & )
* OR(или) ( | )
* XOR(побитовое исключающее или) ( ^ )
* NOT(не) ( ~ )
* LEFT SHIFT(левый сдвиг) ( << )
* RIGHT SHIFT(правый сдвиг) ( >> )
* ZERO-FILL RIGHT SHIFT(правый сдвиг с заполнением нулями) ( >>> )

**Оператор запятая** “,” разделяет выражения, и возвращает результат вычисления последнего из них. Имеет приоритет ниже присваивания

*let* a *=* *(1* *+* *2,* *3* *+* *4);*

alert( a ); // 7 (результат вычисления 3 + 4)

**Операторы сравнения**

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

Операторы сравнения возвращают булевое значение.

При сравнении разных типов JS приводит каждое из них к числу.

|  |  |
| --- | --- |
| True | 1 |
| False | 0 |
| “” | 0 |
| null | 0 |
| undefined | NaN |
|  |  |
|  |  |
|  |  |

Чтобы избежать ошибок при приведении типов используется строгое сравнение (===). Если выражения разных типов, то оно сразу возвращает false.

Особые ситуации:

* Null == undefind //true
* Null == 0 //false (при нестрогом неравенстве для null и undefind действует особое правило – эти значения ни к чему не приводятся. Они равны друг другу и больше ничему)
* NaN при любых сравнениях возвращает false
* Объекты сравниваются по ссылке, true вернется только если сравнивать объект с самим собой.

Логические:

|| - логическое или. Проходит все элементы по очереди, находит первый true и возвращает его. Иначе возвращает последний элемент.

&& - логическое и. Проходит все элементы по очереди, находит первый false и возвращает его. Иначе возвращает последний элемент.

! – логическое не. Сначала приводит аргумент к логическому типу, затем возвращает противоположное значение.

Приоритет && больше чем у ||.

**Оператор нулевого слияния (??)** – возвращает первый аргумент, если он не null/undefind, иначе второй. Он похож на || по действию. Разница в том, что ?? вернет элемент если он равен пустой строке либо нулю. || не вернет их, так как при приведении к bool они будут false.

**ПЕРЕМЕННЫЕ И ФУНКЦИИ**

Переменная представляет именованную область памяти, которая хранит данные. В JS она представляет собой свойство специального объекта – лексического окружения. Лексическое окружение создается для каждого блока кода, заключенного в {};

У цикла для каждой итерации создается свое лексическое окружение.

Для объявления переменной используется ключевое слово **let**.

Имя переменной должно содержать только буквы, цифры или символы $ и \_ и не должно начинаться с цифры. Оно также не должно быть зарезервированным словом.

Чтобы объявить **константу** используется **const**.

При объявлении с помощью let и const переменная видна только внутри блока, где она объявлена – **Блочная область видимости** (внутри скобок {}, допустим в блоке if, либо в цикле).

При объявлении с **var** область видимости переменной ограничивается либо функцией, либо, если переменная глобальная, то скриптом. Var также допускает повторное объявление. Такая переменная будет видна в любом месте функции, независимо от того, где она объявлена. Var не имеет блочной области видимости.

Переменные с var обрабатываются в начале выполнения функции. Такая переменная считается объявленной с самого начала исполнения функции, независимо от того в каком месте функции она была объявлена. Это называется **всплытием (hosting)**. Объявления переменной всплывают, но присваивания нет.

function sayHi() {

alert(phrase);

*var phrase = "Привет";*

}

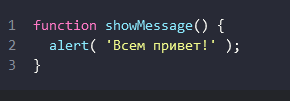
sayHi(); // выведет undefind

**Область видимости модуля:** переменная видна только в том модуле, где она объявлена. При импорте в другом модуле, она будет недоступна, если не указать **export**.

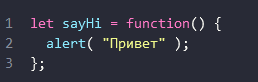
В **Глобальной области видимости** хранятся переменные, доступные в любом модуле. Примером является объект window.

Когда мы обращаемся к переменной, сначала она ищется во внутреннем лексическом окружении, потом во внешнем и т.д. до глобального.

Для создания **функций** мы можем использовать объявление (function declaration)

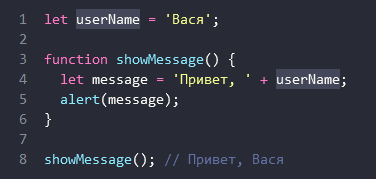


Либо использовать функциональное выражение (Function expression)



Таким образом, функция хранится в переменной. Мы можем присваивать эту функцию другой переменной, передавать как аргумент и возвращать в качестве результата других функций.

Переменные, объявленные внутри функции, видны только внутри этой функции. Функция имеет доступ к внешним переменным, и может изменять их значения.



Можно создавать локальные переменные с таким же именем как и глобальные, тогда обращение будет происходить к локальной переменной.

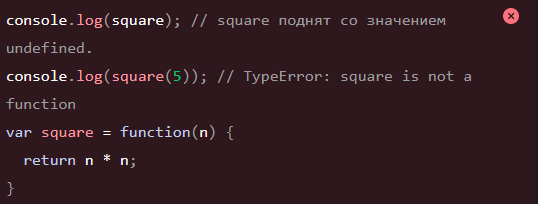
Если при вызове функции не передать один из параметров, он будет undefind. Можно задавать значение по умолчанию

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

Если функция не возвращает значения явно, то возвращается undefind.

Объявления функций function declaration, как и переменных с var, всплывают. Поэтому мы можем вызывать функции до их объявления.

Но такой код не будет работать:



Все функции «при объявлении» получают скрытое свойство **[[Environment]],** которое ссылается на лексическое окружение места, где они были созданы.

Таким образом, при вызове функции, cсылка на внешнее окружение будет устанавливаться в значение [[Environment]]. Это внешнее лексическое окружение будет существовать до тех пор, пока на него будут ссылаться внутренние функции (Даже если внешняя функция уже отработала, и ей окружение больше не нужно).

**Стрелочные функции**



**DOM**

**Document Object Model** (Объектная модель документа) – программный интерфейс для HTML (XML) документов. Это структурированное представление документа в виде дерева объектов, как прослойка между документом и JS.

Каждый HTML-тег представляется в виде объекта. Все эти объекты доступны для JS через глобальный объект **document.**

Типы узлов (всего 12):

* **Элемент (**ELEMENT\_NODE**)** – объект, представляющий html-тег. Элементы образуют древовидную структуру.
* **Текстовый (**TEXT\_NODE**)** – текст внутри элементов (тегов). У него не может быть потомков.
* **Комментарий (**COMMENT\_NODE**)** (иногда в них можно включить информацию, которая не будет отображаться браузером, но будет доступна через DOM для чтения JS)
* **Узел инструкции обработки (**PROCESSING\_INSTRUCTION\_NODE**) –** в xml используется для передачи информации приложениям. 
* **Узел документа** (DOCUMENT\_NODE) - сам document, корневой элемент DOM
* **Узел типа документа (**DOCUMENT\_TYPE\_NODE**) (**<!DOCTYPE>**)**
* **Узел фрагмента документа (**DOCUMENT\_FRAGMENT\_NODE) – «облегченный» объект Document, может содержать часть документа.
* **Узел аттрибута**(ATTRIBUTE\_NODE)
* **Узел CDATA**(CDATA\_SECTION\_NODE)
* (NOTATION\_NODE) – нотация, объявленная в DTD
* **Узел сущность (**ENTITY\_NODE**) –** какая-то сущность
* **Узел ссылка на сущность (**ENTITY\_REFERENCE\_NODE**) –** ссылка на сущность

Dom-коллекции доступны только для чтения. Чтобы изменить их, нужно пользоваться специальными функциями.

Они также динамически обновляются, при обновлении dom и html-документа.

Для их перебора нужно использовать цикл **for..of (**for..in перебирает все свойства коллекции, а нам нужны только дочерние элементы**)**.

Самые верхние элементы дерева доступны как свойства объекта document.

* document.**documentElement** – самый верхний узел документа, соответствует тегу <html>.
* document.**body**
* document.**head**

Свойства и методы элементов для навигации по узлам:

* **childNodes –** список дочерних узлов (дочерние=непосредственно вложенные в него)
* **firstChild** – первый дочерний узел
* **lastChild –** последний дочерний узел
* **hasChildNodes() –** метод для проверки, есть ли у элемента дочерние узлы
* **nextSibling** – следующий сосед (узел на том же уровне вложенности)
* **previousSibling** – предыдущий сосед
* **parentNode** – родительский узел

Свойства для навигации **Только по узлам-элементам** (без текстовых, комментов и т.д.)

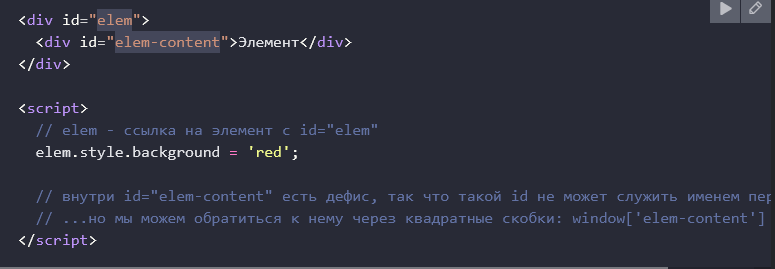
* **children** – коллекия дочерних элементов
* **firstElementChild**, **lastElementChild** – первый и последний дочерний элемент
* **previousElementSibling**, **nextElementSibling** – соседи-элементы.
* **parentElement** – родитель-элемент (parentNode и parentElement возвращают практически всегда одно и тоже, за исключением когда вызываются на элементе *document.documentElement*. parentNode вернет document, а parentElement - null)

Некоторые элементы предоставляют дополнительные ссылки и коллекции для доступа к своему содержимому.

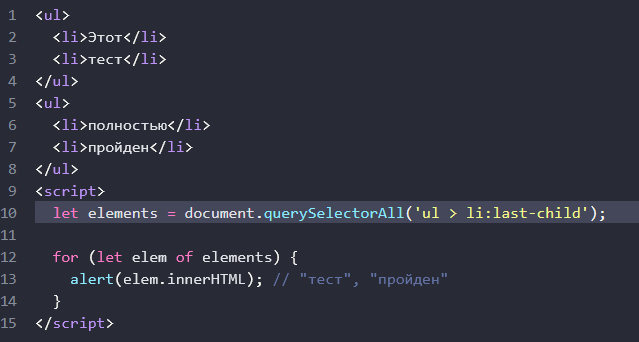
Свойства для **получения произвольных элементов** страницы.

document.**getElementById(id) –** получить элемент по его свойству id.

Если id не уникален, то поведение непредсказуемо (но обычно в браузерах реализовано получение первого такого элемента).

Так же существует глобальная переменная с именем, указанным в id (но лучше такое не юзать. Могут появиться конфликты имен, и при чтении js кода непонятно откуда берется переменная, и что это вообще такое).  


**elem.querySelectorAll(css)** – самый универсальный метод, возвращает все элементы внутри elem, удовлетворяющее данному css-селектору.



**elem.querySelector(css)**  - возвращает первый элемент, соответствующий данному селектору.

**elem.matches(css)** – проверяет, удовлетворяет ли элемент заданному селектору.

**elem.closest(css)** – ближайший предок, который удовлетворяет селектору.

Методы **getElementsBy\*:**

* **elem.getElementsByTagName(tag) –** ищет вложенные элементы, с указанным тегом. Передав в параметр “\*” можно получить всех потомков.
* **elem.getElementsByClassName(className)** – возвращает элементы, которые имеют свойство class равное переданной строке.
* **document**.getElementsByName(name) – возвращает элементы с заданным атрибутом name.

Все методы "getElementsBy\*" **возвращают «живую» коллекцию**. Они автоматически обновляются при изменении документа.

querySelectorAll возвращает **статическую** коллекцию.

**BOM**

Browser Object Model – это дополнительные объекты, предоставляемые (окружением) для взаимодействия с браузером.

Главным является объект **window**. Он представляет собой окно браузера (вкладку). Для каждой вкладки свой объект window. Все глобальные переменные и функции являются свойствами этого объекта.

Свойства объекта window (если работаем с текущим окном, то window можно опускать):

* **window.innerHeight** – внутренняя высота окна браузера (пиксели)
* **window**.**innerWidth –** внутренняя ширина окна браузера (пиксели). Окно браузера не включает панель инструментов и полосу прокрутки.
* **window**.**open([url]) –** открывает новое окно браузера.
* **window.close**() - закрывает текущее окно
* **window.moveTo**() - передвигает текущее окно
* **window.resizeTo()** - изменяет размер текущего окна
* **window**.**alert() –** всплывающее окно для вывода сообщений
* **window**.**confirm() –** окно, в котором пользователю необходимо подтвердить или отменить действие
* **window**.**prompt() –** окно для получения данных от пользователей

**screen** – объект, который предоставляет информацию об экране пользователя:

* **screen.width –** возвращает ширину экрана пользователя (пиксели)
* **screen.height –** высота экрана пользователя (пиксели)
* **screen.availHeight –** доступная высота экрана(без панели задач и т. п.)
* **screen.availWidth –** доступная ширина экрана
* **screen.colorDepth –** число бит, используемых для отображения цвета. (глубина цвета). Обычно 24 или 32.

**navigator –** информация о браузере пользователя. Информация из объекта navigator может вводить в заблуждение! Не следует использовать ее чтобы узнать версию браузера и т.п. Свойства:

* **navigator.appName –** возвращает имя браузера как приложения. именем приложения для браузеров IE11, Chrome, Firefox и Safari является "Netscape".
* **navigator.appCodeName –** возвращает кодовое имя браузера. Кодовым именем для браузеров Chrome, Firefox, IE, Safari и Opera является "Mozilla"
* **navigator.product –** возвращает имя движка браузера. Не стоит полностью полагаться на данные этого свойства. Большинство браузеров возвращают имя "Gecko"!!
* **navigator.appVersion –** информация о версии браузера.
* **navigator.** **userAgent -** заголовок пользовательского агента, который браузер отправляет серверу
* **navigator.platform –** платформа браузера (ОС)
* **navigator.language** – язык браузера.

**history –** объект, позволяющий получить историю переходов пользователя по ссылкам в пределах одного окна браузера. Данный объект отвечает за кнопки вперед, назад. С помощью него можно переходить вперед назад.

* **history.back() –** перейти на предыдущий url в списке посещенных страниц
* **history.forward() –** загружает следующий url в списке посещенных страниц.
* **History.lenght() –** доступный размер истории
* **History.go(число) –** перейти на указанное кол-во страниц в истории

**location** – объект, отвечающий за адресную строку браузера.

* **location.href** – возвращает url текущей страницы.
* **location**.**hostname –** возвращает имя хоста текущей страницы
* **location**.**pathname –** возвращает путь к текущей странице
* **location**.**protocol –** возвращает протокол веб-страницы (http, https)
* **location**.**port –** возвращает номер порта (большинство браузеров не показывают номер порта по умолчанию 80-http, 443-https)
* **location**.**assign(url) –** загружает новый документ по url

**НАЗНАЧЕНИЕ ОБРАБОТЧИКОВ СОБЫТИЙ**

Событие – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы (и не только Dom).

Часто используемые события DOM:

**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.

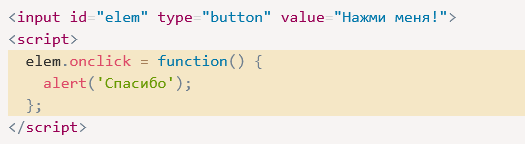
Событию можно назначить **обработчик**, то есть функцию, которая сработает, как только событие произошло. Таким образом, JS-код может реагировать на действия пользователя.

**Назначение обработчиков**

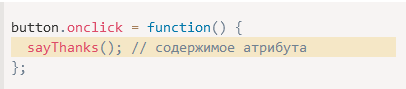
**прямо в разметке**, в атрибуте **on<событие>**:



**Свойство DOM-объекта, on<событие>**:



Если обработчик задан через атрибут, то браузер читает HTML-разметку, создаёт новую функцию из содержимого атрибута и записывает в свойство.



Этот способ, по сути, аналогичен предыдущему. Обработчик всегда хранится в свойстве DOM-объекта, а атрибут – лишь один из способов его инициализации.

Так как у Элемента DOM может быть только одно свойство с определенным именем, то **таким способом** **назначить более одного обработчика нельзя**.

**addEventListener** – позволяет назначать несколько обработчиков.



**event** – имя события, например “click”;

**handler –** ссылка на функцию обработчик;

**options** - Дополнительный объект со свойствами:

* **once**: если true, тогда обработчик будет автоматически удалён после выполнения.
* **capture**: фаза, на которой должен сработать обработчик. Так исторически сложилось, что options может быть false/true, это то же самое, что {capture: false/true}. Если false (по умолчанию), событие будет поймано при всплытии. True – при погружении.
* **passive**: если true, то указывает, что обработчик никогда не вызовет preventDefault(). Есть некоторые события, как touchmove на мобильных устройствах, которое по умолчанию начинает прокрутку, но мы можем отменить это действие в обработчике. Браузер запустит прокрутку только после выполнения всех обработчиков, если никакой обработчик ее не отменит. Указав passive: true мы сообщим браузеру, что не собираемся отменять обработку по умолчанию, и браузер сразу начнет прокрутку.

Для удаления обработчика следует использовать **removeEventListener**



Для удаления нужно передать именно ту функцию, которая была назначена. Если функцию обработчик не сохранить, мы не сможем ее удалить. Нет метода, который позволяет получить из элемента обработчики событий, назначенные через addEventListener.

Есть события, которые можно назначить только через addEventListener например DOMContentLoaded.

Внутри обработчика события this ссылается на элемент, на котором висит обработчик.

Обработчики вызываются в порядке их назначения.

**Объект-обработчик** также может быть назначен в качестве обработчика с помощью addEventListener. В этом случае, когда происходит событие, вызывается метод объекта **handleEvent**.

**Объект события –** объект, который содержит детали события. Браузер создает его, когда происходит событие, и передает его в качестве аргумента функции обработчику.

Свойства:

* event.type – тип события
* event.currentTarget – (= this) элемент до которого дошло всплытие, на нем сейчас выполняется обработчик.
* event.target – целевой элемент, на котором произошло событие.
* event.eventPhase – номер фазы, на которой поймано событие.
* event.clientX / event.clientY – координаты курсора в момент клика относительно окна, для событий мыши.

target – объект, на котором произошло событие

current-target – объект на который наложен обработчик

**3 фазы прохода события**

**Фаза погружения** – событие идет сверху вниз

**Фаза цели** – событие достигло целевого (исходного) элемента.

**Всплытие -** событие начинает всплывать.  
Когда на элементе происходит событие, обработчики сначала срабатывают на нем, потом на его родителе и так далее, вверх по цепочке предков до объекта document(иногда window). Всплывают не все события (focus).

Любой промежуточный обработчик может остановить всплытие, вызвав *event.stopPropagation()*

Если у элемента есть несколько обработчиков, то даже при прекращении всплытия будут выполнены все обработчики на текущем элементе. Метод stopImmediatePropagation() предотвращает всплытие и останавливает обработку событий на текущем элементе.

**Делегирование событий**

Идея в том, что если у нас есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы назначать обработчик каждому, мы ставим один обработчик на их общего предка.

В нем мы можем получить целевой элемент event.target, чтобы понять, на каком элементе произошло событие.

**Отмена действий браузера** нужна чтобы отменить дефолтное поведение браузера при возникновении события (переход на новый url при клике на ссылку, отправка формы при нажатии на кнопку отправить, выделение текста при зажатии кнопки мыши.)

* в обработчике можно вызвать **event.preventDefault();**
* если обработчик назначен через on<событие>, то также можно вернуть false из обработчика.

**ДОП ВОПРОСЫ**

**Как отличть массив от объекта?**

Воспользоваться методом **Array.isArray(obj)**