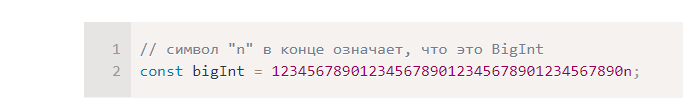
**Данные**

* **Какие типы данных есть в JS?**

Переменная в JS может содержать любой тип данных и переопределяться. Такой язык называется «динамически типизированным».

В JS **8 типов данныx**:

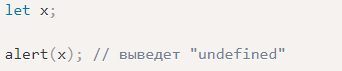
**Число –** числовой тип(числа и числа с плавающей точкой)  
**BigInt –** для работы с очень большими числами (которые больше чем 2^53)



**Строка –** для работы со “строками”  
**Булевый тип** – логический тип данных (true и false)  
**null** – ничего, пусто, значение не известно.



**Undefined**  - значение не присвоено. Если есть переменная, но в ней нет никакого значения.

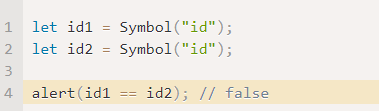


**Объект** – для хранения коллекций данных или более сложных объектов.

**Symbol**(символ)– cоздание уникальных идентификаторов объекта.

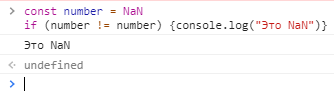
* **Какие типы в JS изменяемые, а какие – нет?**

Неизменяемый тип данных – Symbol. Символ гарантированно уникальный идентификатор.

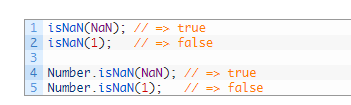


* **Что означает ссылка на переменную?**Переменная – это «именовоное» хранилище для различных данных. Ссылка на переменную это обращение к этой переменной.
* **Что такое NaN? Как проверить, что переменная NaN?**

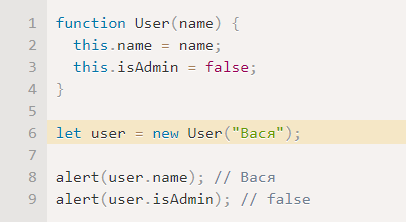
Not-A-Number. Означает «не число». NaN относится к числовому типу данных. Оно не равно ничему, даже самому себе. Поэтому, чтобы проверить является ли переменная NaN, можно просто сравнить её саму с собой



JavaScript содержит встроенные функции для определения NaN: isNaN() и Number.isNaN()



* **Что означает создавать переменную через конструктор или литерал?**
* Литерал – это любое значение указанное явным образом в коде. В качестве них могут быть числа, строки, логические значения. Создание через конструктор:



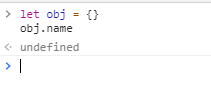
* **Когда при обращении к объекту использовать точечную нотацию, а когда через строку в квадратных скобках?**

Для обращения к свойству объекта необходимо использовать точку (свойство.значение).  
Точка требует, чтобы ключ был именован по правилам именования переменных.

[Квадратные скобки] это альтернативный доступ к свойству, который позволяет обойти правила именования переменных:

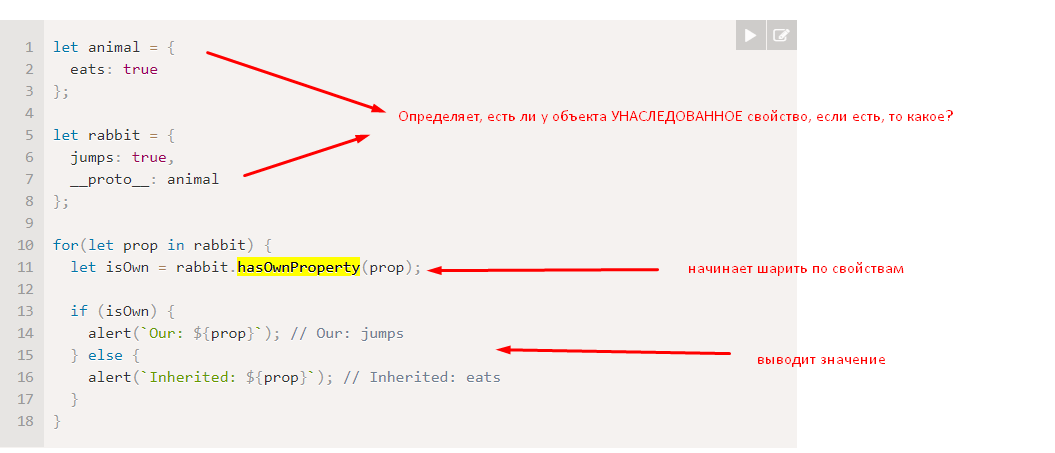


* **Что произойдет, если попытаться получить несуществующее в объекте свойство?**



* **Что делает hasOwnProperty?**

Это относится к прототипам. Этот метод нужен, чтобы отфильтровать унаследованные свойства (отбросить унаследованное, если есть своё). Вернет true, если у объекта есть собственное не унаследованное имя



* **Когда использовать null, а когда undefined?**

Они оба абсолютно не способны измениться, не имеют свойств и методов.

Любая объявленная переменная без значения будет undefined. Undefined так же возникает при попытке обратиться к несуществующему свойству объекта или массива, а так же, если функция п принимает несуществующий аргумент, то он тоже undefined.

Null отличается своим применением и больше используется для присваивания значения. Если нужно присвоить «не значение» переменной или свойству, то null хороший вариант. Так же присвоив объекту null мы тем самым отключим его, удалив его ссылку, и если на него нет других, он уйдет к сборщику мусора и освободит память.

* Какие есть способы создания глобальных переменных?

Глобальная переменная, это та переменная, которая объявлена вне тела функции. Все глобальные переменные являются свойствами глобального объекта – window(в браузере. В ноде это global).  
Если переменная объявлена вне функции – она глобальная.  
Важное свойство можно записать напрямую windows.globalMyLet = “тест” – оно глобальное

Но так делать не рекомендуется, все переменные должны быть локальными.

* Для чего нужна деректива “use strict”?

Это строгий режим, введен в ES5.

Может включаться не только глобально, но и для конкретных функций.

Он по умолчанию включен в консоли браузера.

Некоторые функции языка (Классы и модули) автоматически включают строгий режим.

Он убивает программу на некоторых ошибках, которые интерпретатор раньше просто пропускал. Он запрещает использовать некоторые элементы синтаксиса, давать одинаковые названия свойствам и аргументам функции, запрещает восьмиричную систему исчисления и тд.

**Выражения**

* Что такое выражения и инструкции? В чем отличие между ними?

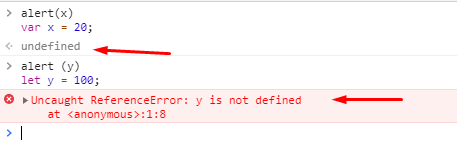
**Инструкция** – это указание на совершение какого-либо действия, например, создать переменную, запустить цикл, выполнить условную инструкцию, выйти из функции и т. п. Любая программа представляет собой последовательность выполняемых инструкций. Окончание инструкции обозначается символом ; (точка с запятой).

**Выражение** – это любая часть исходного кода программы, которая вычисляет значение и больше ничего не делает: она никак не изменяет состояние программы.

* Чем отличаются var, let и const? Почему использование const может быть предпочтительней?

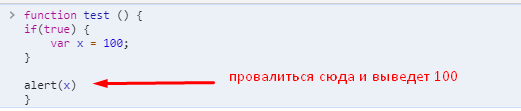
**let и const**, в отличии от **var**:  
-видна только в рамках блока, в котором была объявлена {}

-становится видимой после своего объявления (если var не объявлена, она undefined)

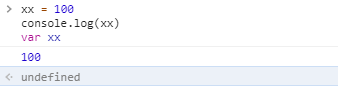


-нельзя объявлять переменную повторно

**var**:  
-нет блочной видимости, он просто игнорирует {}  
-область видимости, в пределах функции (то есть может выходить из других блоков):



-var сразу считается объявленной в момент старта функции вне зависимости от того, в каком месте она реально находится:



**Const**:

-создает констнанту, которую нельзя изменить.

(если в константу присвоен ОБЪЕКТ), то нельзя поменять сам ОБЪЕКТ, но свойства внутри него – можно)

* Что такое тернарный оператор?

Он же называется условный оператор - ?

Его синтаксис: let box = УСЛОВИЕ ? true: false

* Что делает оператор for…in? Особенности его использования с массивами?

Цикл проходит через каждое свойство объекта, по каждому элементу.   
Для массивов тоже можно использовать его, т.к массив тоже своего рода объект, но с тем условием, что нет гарантии, что индекса массива вернуться в правильном порядке. Поэтому лучше использовать для массивов for…of.

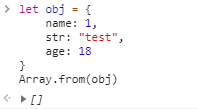
* Как проверить существует ли переменная и не поймать ошибку ReferenceError?

typeof variable != undefined;

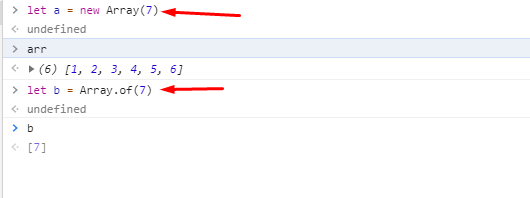
**Массивы**

* Способы создания массивов (литерал, конструктор, фабричные методы - Array.from() и Array.of()

Let arr = []; - литерал  
let arr = new Array(); - конструктор   
Array.from – принимает итерируемый объект и делает из него массив



Array.of – создает новый массив Array из произвольного числа аргументов.  
Разница от простого Array в ЦЕЛОЧИСЛЕННОЙ обработке аргументов:

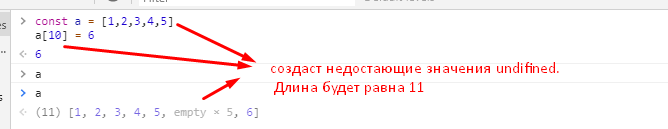


* **Удаление элеменнтов из массива (какие способы есть, в чем отличие?)  
  splice**(index,[, deleteCount, elem1 …., elemN]) – добавляет, удаляет и изменяет элементы.

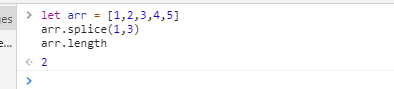
Начинает с позиции Index удаляет deleteCount и вставляет elem 1,…. И возвращает результат работы. Меняет массив.

Pop() – извлекает элементы с конца  
shift() – извлекает элементы с начала

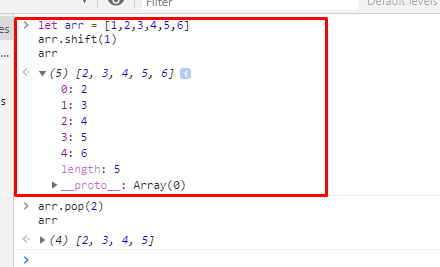
* Свойство length у массива. Какие способы удаления элементов массива влияют на длину, а какие не влияют?  
  length – длинна массива.



Мутируют массив методы splice:

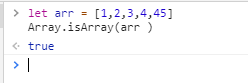
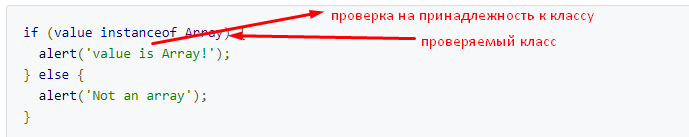


Метод shift и pop тоже его мутируют:



* Как проверить, что в переменной лежит массив?  
  один из способов:

более элегентный способ:



* Что делают, как и когда использовать следующие методы:

**Sort()**

Сортирует массив на месте, от меньшего к большему, приводя значения к строкам.

В аргументы метода нужно передавать функцию, которая определяет порядок сортировки. Функцию над чиркануть самому.

**Filter()**

Возвращает массив из всех подходящих элементов.

Вернет пустой массив в случае, если ничего не найдено.

**ForEach(перебор)**

Запускает функцию для каждого элемента массива.

**Map()**

Вызывает функцию для каждого элемента массива и возвращает значение этой функции. Его отличает от других как раз то, что он генерирует новый массив на основе старого.

**Some()**

Проверяет, удовлетворяет ли какой-либо элемент массива условию, заданному в функции.

**Every()**

Проверяет, удовлетворяют ли все элементы массива условию, заданному в передаваемой функции.

**Reduce()**

Цель этого метода – взять массив и сжать его содержимое до одного значения. Это может быть строкой, числом, объектом или другим массивом. Очень гибкий метод.

Принимает два аргумента – калбек для каждого элемента в массиве и начальное значение.

Калбек тоже принимает два аргумента – аккумулятор(текущее объединенное значение) и текущий элемент в цикле.   
Все, что мы возвращаем, используется в качестве аккумулятора для следующего элемента в цикле. В самом первом цикле используется начальное значение.

Его смысл в том, что с помощью него мы можем вычесть какое-либо единое значение на основе всего массива.

**Функции**

* Какие есть 4 шаблона вызова функции, которые задают контекст выполнения этой функции? Как директива ‘use string’ влияет на this внутри функции?

1.Вызов метода – method Invocation

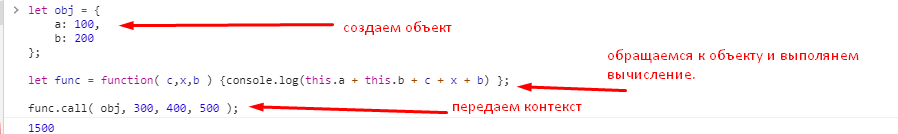
2.Вызов функции – Function Invocation

3.Вызов конструктора – Construction Invocation

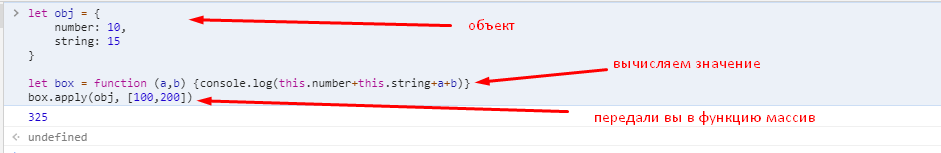
4.Вызов apply и call

Функции очень гибкие, их можно использовать в других функциях и объектах.

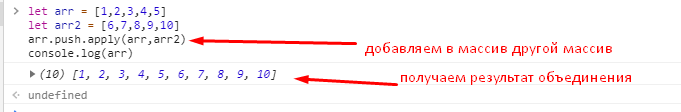
Метод **call** позволяет вызывать функцию как метод объекта, устанавливая ей this и передавая необходимые аргументы, а можно вообще обойтись без их передачи:



**Apply** похож на метод call, также позволяет передать функцию как метод объекта, но передает аргументы в виде массива:



Можно также соединить два массива:



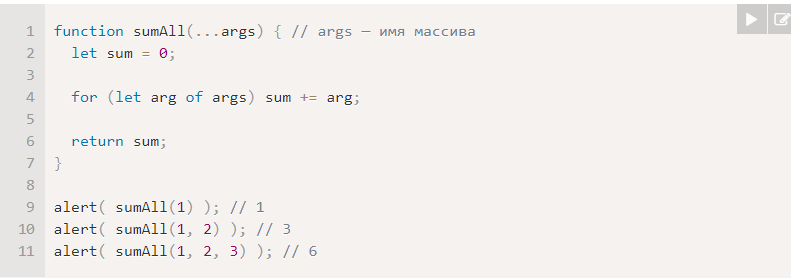
**This** в функциях которые запущены в строгом режиме = undefined. И во всех вложенных в нее функциях – тоже.

* **Какой наиболее простой паттерн облегчает читаемость функции, когда у нее огромное количество аргументов?**

Можно использовать оператор Rest для упаковки аргументов.

* **Как получить все аргументы функции, которые были не объявлены, но все таки были переданы?**

Вызвать функцию можно с любым количеством аргументов. Можно собрать оставшиеся параметры через три точки … и положить их в массив  
Так же имеется старый метод, по которому можно обратиться к аргументам функции. Все аргументы у функции находятся в псевдомассиве arguments:



* Что такое рекурсия? Когда её удобно использовать?

Рекурсия – вызов функцией самой себя.

Для рекурсии необходим:  
1)Базовый случай (остановка)

2) Правило для углубления (для повторного запуска)

Что такое замыкания, когда они могут быть полезны? Как сохранить состояние с помощью замыкания и для чего это состояние может быть использовано?

Замыкание, по сути, можно понимать как «замкнуть в себе что то», в примере выше функция test замкнула внутри себя анонимную функцию с входящей переменной n, и чтобы получить до неё доступ мы создали переменную и записали в неё функцию.   
Как этим пользоваться?  
  
Мы берем функцию, присваиваем её в переменную(получается другая функция, как мы выяснили выше) и затем работаем с ней, а её текущее значение, которое мы присвоили в замыкание остается неизменным:  
  
Вот рабочий пример для замыканий, который в принципе передает всю их суть:  
  
Собственный метод bind:

