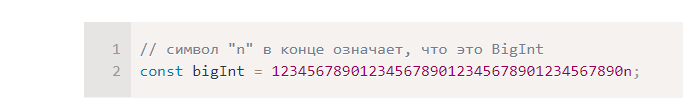
**Данные**

* **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**:

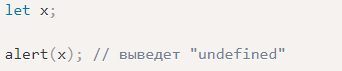
**Число –** числовой тип(числа и числа с плавающей точкой)  
**BigInt –** для работы с очень большими числами (которые больше чем 2^53)



**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.

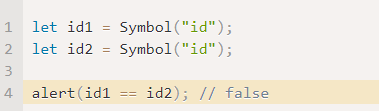


**Объект** – для хранения коллекций данных или более сложных объектов.

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

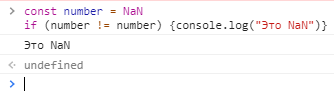
* **Какие типы в JS изменяемые, а какие – нет?**

Неизменяемый тип данных – Symbol. Символ гарантированно уникальный идентификатор.

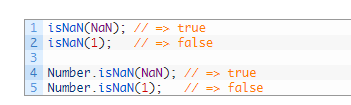


* **Что означает ссылка на переменную?**Переменная – это «именовоное» хранилище для различных данных. Ссылка на переменную это обращение к этой переменной.
* **Что такое NaN? Как проверить, что переменная NaN?**

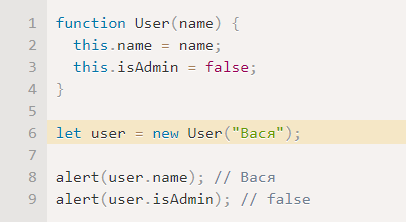
Not-A-Number. Означает «не число». NaN относится к числовому типу данных. Оно не равно ничему, даже самому себе. Поэтому, чтобы проверить является ли переменная NaN, можно просто сравнить её саму с собой



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:



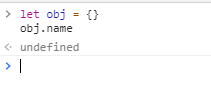
* **Когда при обращении к объекту использовать точечную нотацию, а когда через строку в квадратных скобках?**

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:

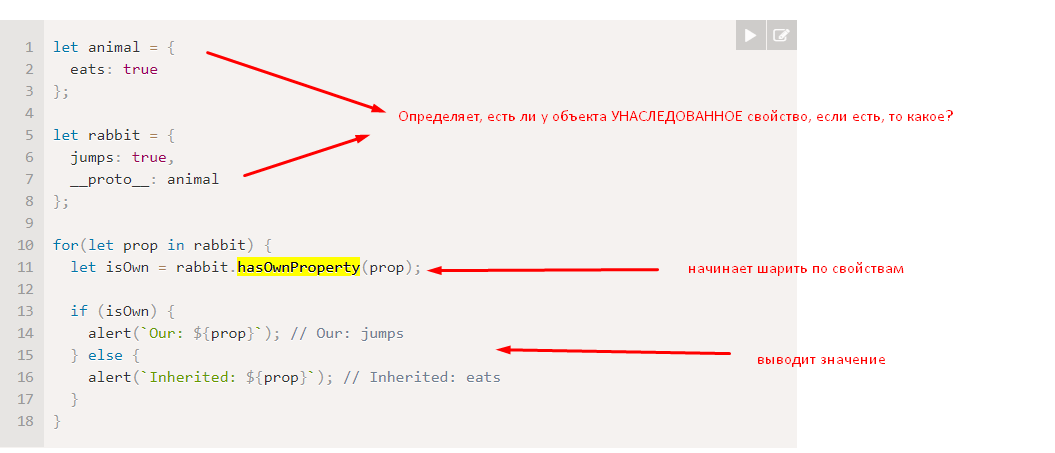


* **Что произойдет, если попытаться получить несуществующее в объекте свойство?**



* **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?

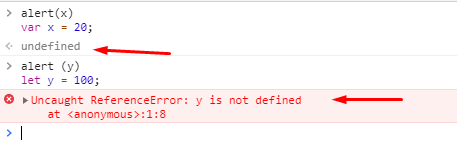
**Инструкция** – это указание на совершение какого-либо действия, например, создать переменную, запустить цикл, выполнить условную инструкцию, выйти из функции и т. п. Любая программа представляет собой последовательность выполняемых инструкций. Окончание инструкции обозначается символом ; (точка с запятой).

**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?

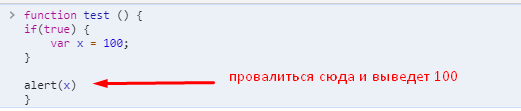
**let и const**, в отличии от **var**:  
-видна только в рамках блока, в котором была объявлена {}

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

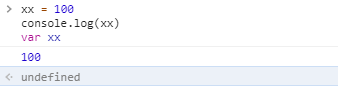


-нельзя объявлять переменную повторно

**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.

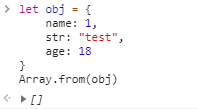
* Как проверить существует ли переменная и не поймать ошибку ReferenceError?

typeof variable != undefined;

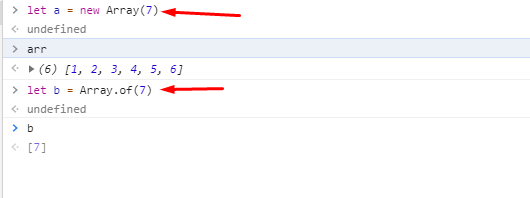
**Массивы**

* 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:

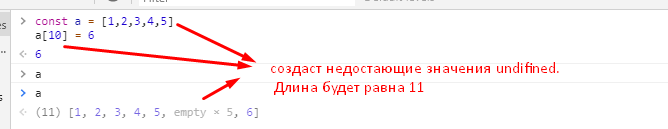


* **Удаление элеменнтов из массива (какие способы есть, в чем отличие?)  
  splice**(index,[, deleteCount, elem1 …., elemN]) – добавляет, удаляет и изменяет элементы.

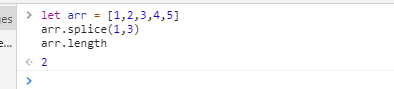
Начинает с позиции Index удаляет deleteCount и вставляет elem 1,…. И возвращает результат работы. Меняет массив.

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

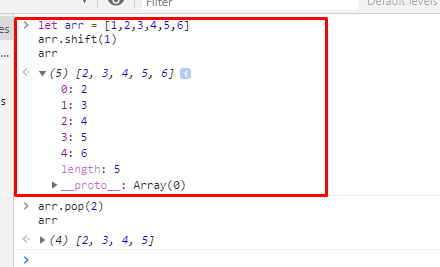
* Свойство length у массива. Какие способы удаления элементов массива влияют на длину, а какие не влияют?  
  length – длинна массива.



Мутируют массив методы splice:

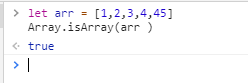
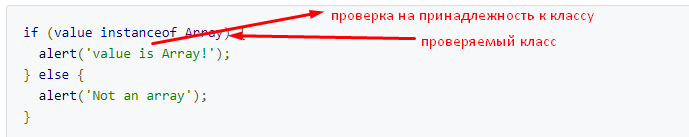


Метод shift и pop тоже его мутируют:



* 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

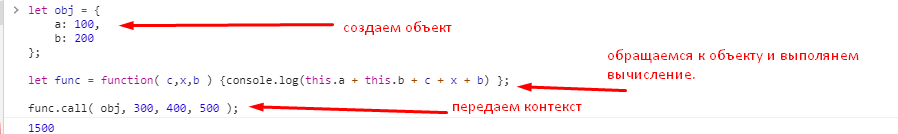
2.Вызов функции – Function Invocation

3.Вызов конструктора – Construction Invocation

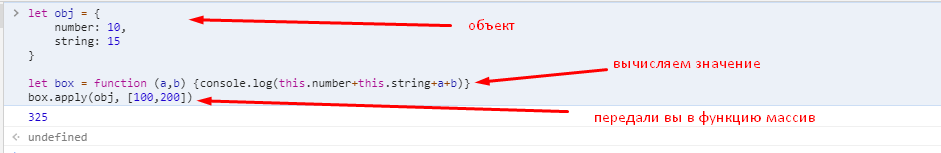
4.Вызов apply и call

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

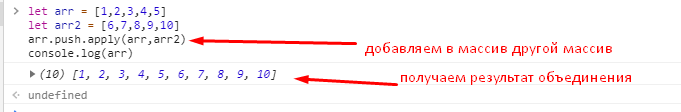
Метод **call** позволяет вызывать функцию как метод объекта, устанавливая ей this и передавая необходимые аргументы, а можно вообще обойтись без их передачи:



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



Можно также соединить два массива:



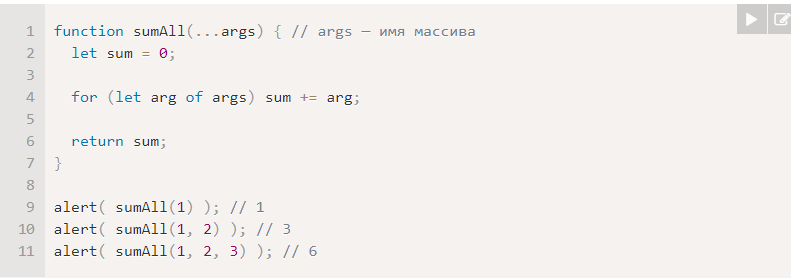
**This** в функциях которые запущены в строгом режиме = undefined. И во всех вложенных в нее функциях – тоже.

* **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.

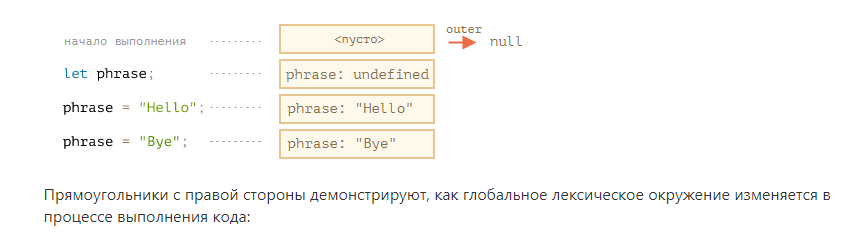
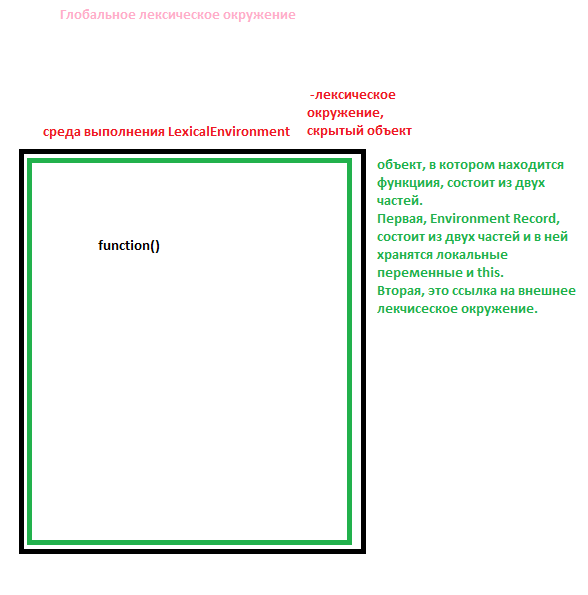
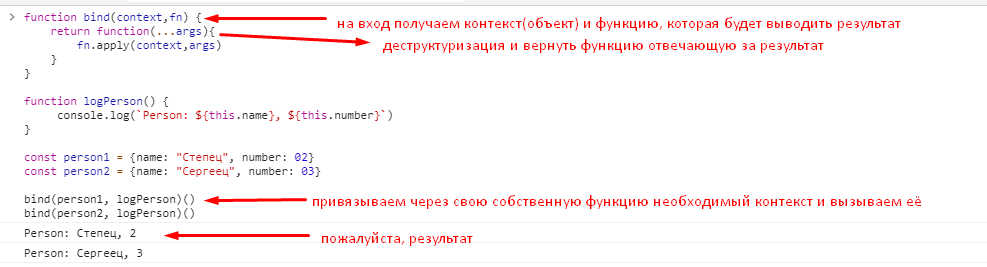
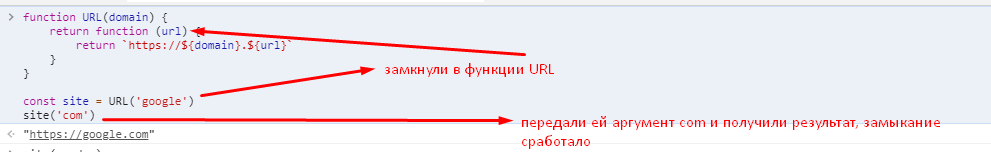
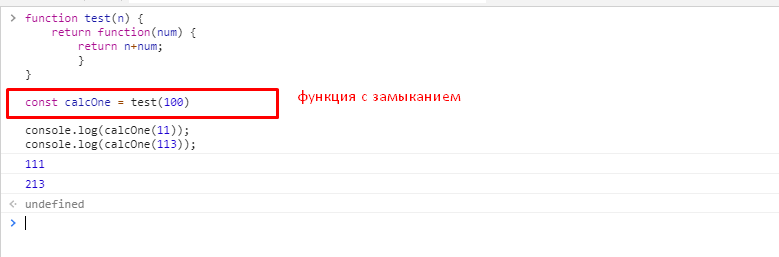
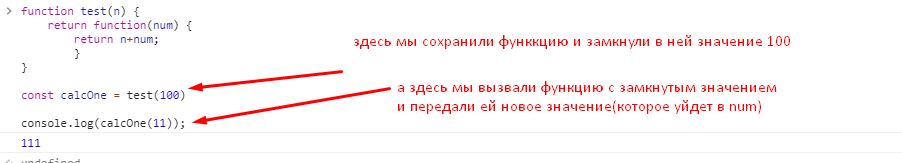
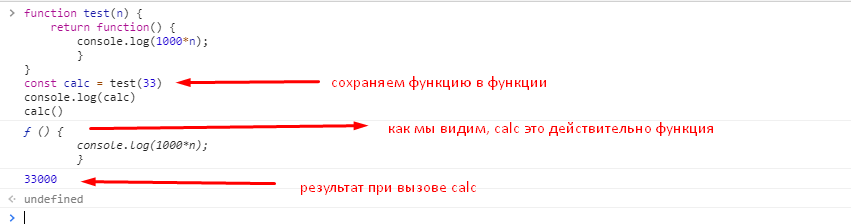
Для рекурсии необходим:  
1)Базовый случай (остановка)

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?

Сначала очень кратенько:

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:  
  
**Более детально о замыканиях и работе функций:**  
**"Переменная" – это просто свойство специального внутреннего объекта: Environment Record. «Получить или изменить переменную», означает, «получить или изменить свойство этого объекта».**

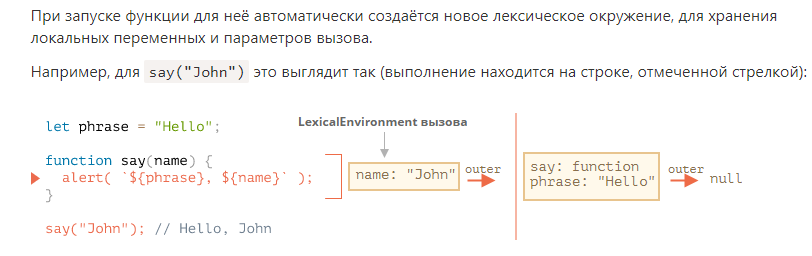
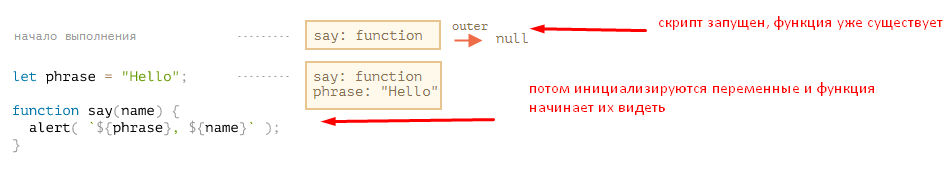


Переменная – это свойство специального внутреннего объекта, связанного с текущим выполняющимся блоком/функцией/скриптом.

Работа с переменными – это на самом деле работа со свойствами этого объекта.

**Функции**Если переменная инициализируется в момент старта лексического окружения, то функция инициализируется в момент выполнения скрипта.

Итак, в процессе вызова функции у нас есть два лексических окружения: внутреннее (для вызываемой функции) и внешнее (глобальное):



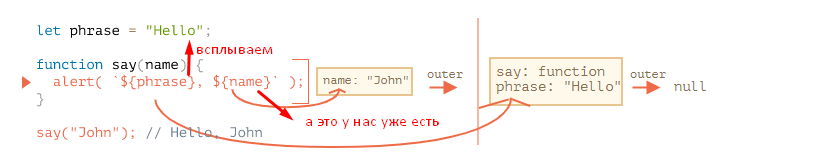
-внутреннее лексическое окружение соответствует текущему выполнению say.

-В нём находится одна переменная name, аргумент функции. Мы вызываем say("John"), так что значение переменной name равно "John".

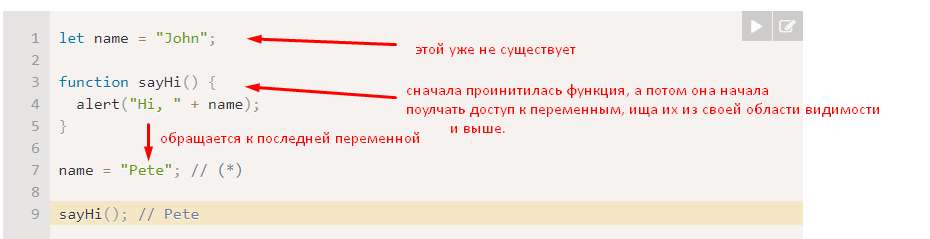
-Внешнее лексическое окружение – это глобальное лексическое окружение.

-В нём находятся переменная phrase и сама функция.

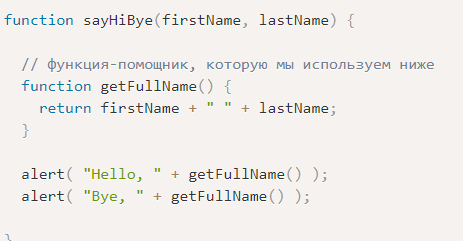
**Когда код хочет получить доступ к переменной – сначала происходит поиск во внутреннем лексическом окружении, затем во внешнем, затем в следующем и так далее, до глобального.**



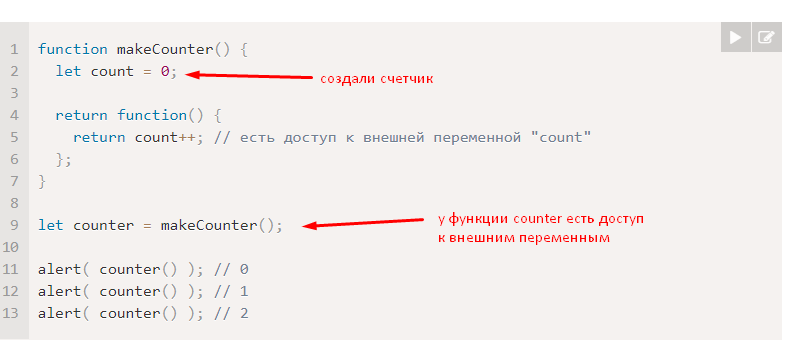
**Функция получает текущее значение внешних переменных, то есть, их последнее значение.**



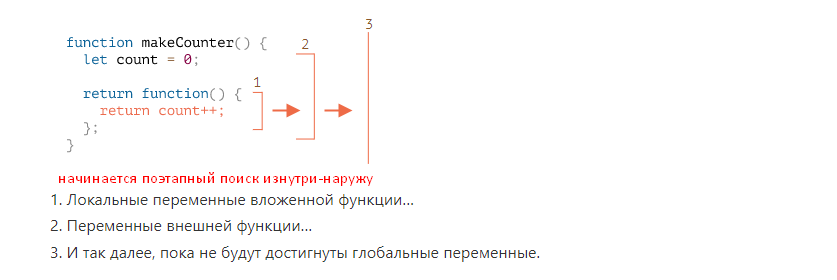
Каждый новый запуск функции – это все время новый поиск нового лексического окружения, то есть если оно меняется каждый раз при вызове функции, функция тоже будет меняться, она динамическая.  
Ненужные переменные вычислит движок по своим алгоритмам и их убьет сборщик мусора.  
  
Функция называется вложенной, когда она создана внутри другой функции. Ниже представлена такая функция помощник:



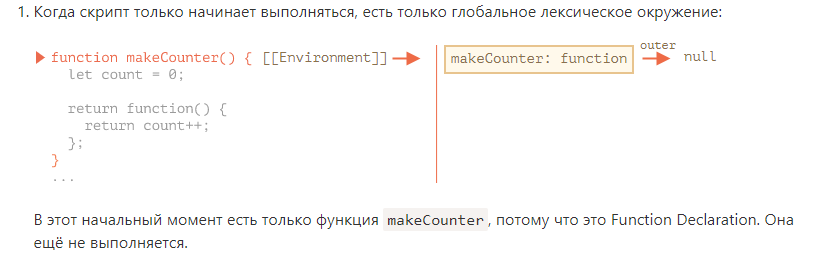
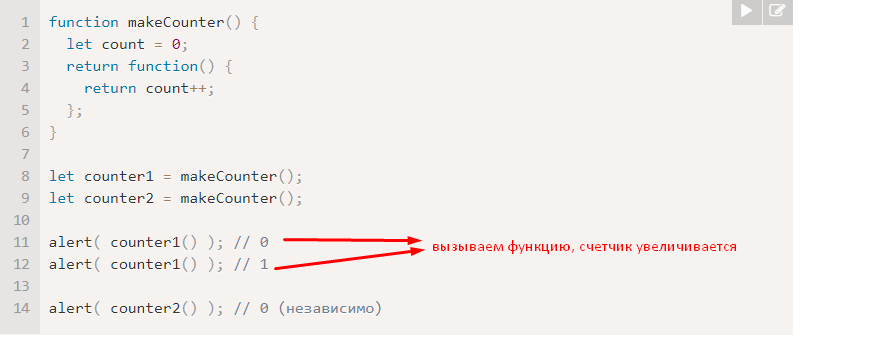
Вложеная функция может быть возвращена и использоваться в любом месте кода. Она будет иметь доступ к внешним переменным.



И каждый такой вызов функции инитит её заново, и все время в ней инитятся новые независимые переменные.



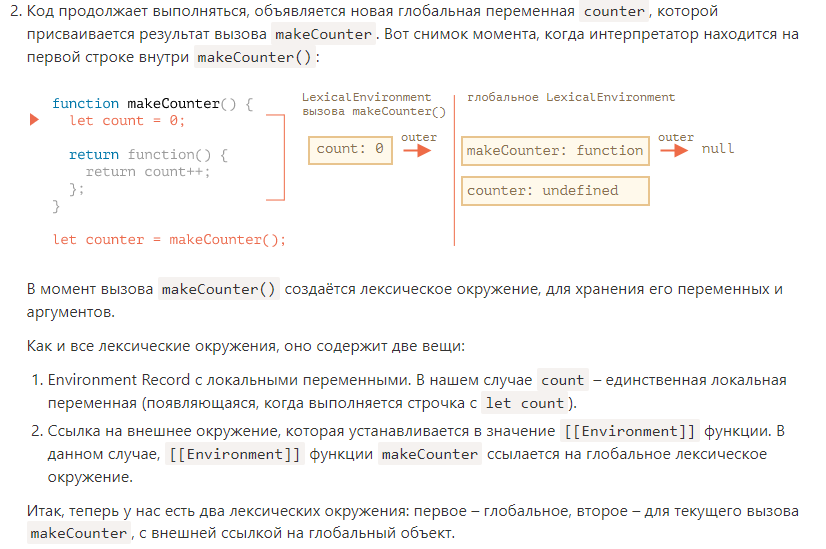
Разберем вызов функции детально, **ШАГ 1**



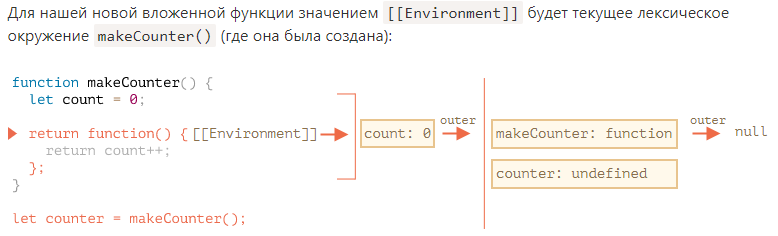
**Все функции «при рождении» получают скрытое свойство [[Environment]], которое ссылается на лексическое окружение места, где они были созданы.  
И это важно!**

Это как раз то, откуда функция знает про место своего рождения. [[Environment]] специальное скрытое свойство у функции, которое содержит ссылку на место её рождения.

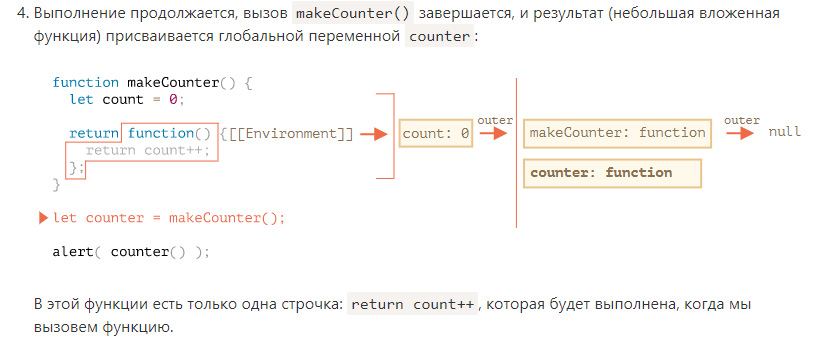
**Шаг 2**



**Шаг 3**Здесь в процессе выполнения создается маленькая вложенная функция. Причем не важно, какая это функция обычная(декларативная) или стрелочная(Expression), они обе все равно получают скрытое свойство Environment, которое будет ссылаться на лексическое окружение, их породившее.

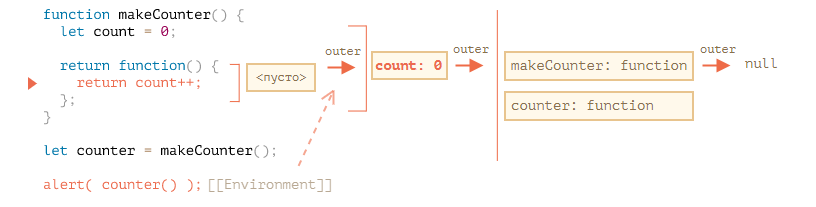


Интерпретатор идет по шагам и код выше ещё не выполнен, поэтому counter не определена.  
А здесь код выполнился:  
**Шаг 4**

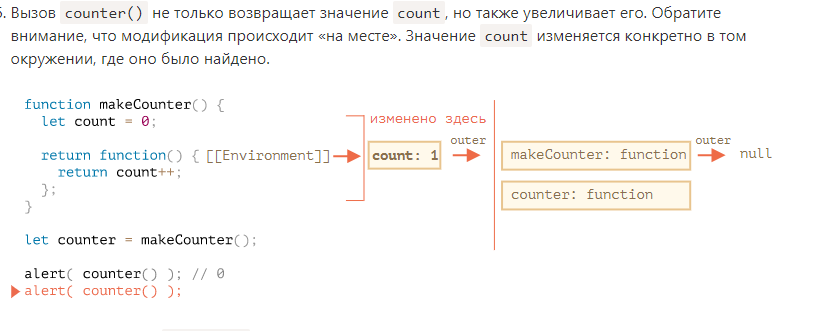


**Шаг 5**

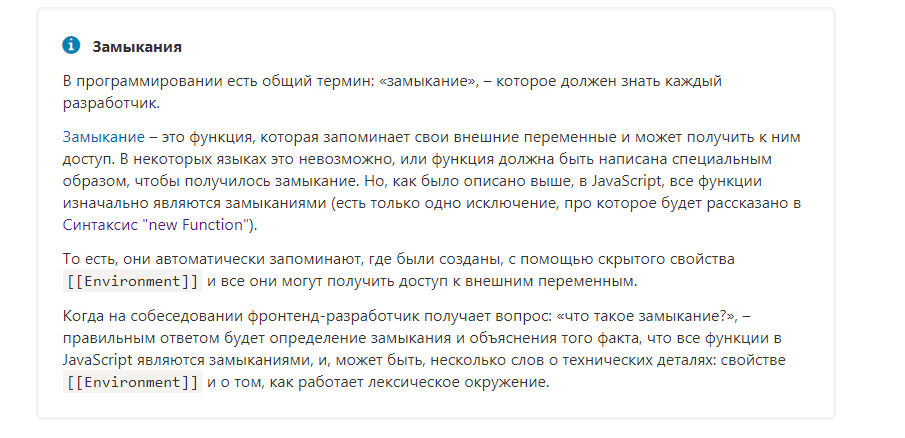
При вызове counter (функции, которой мы присвоили другую функцию) для него создается новое лексическое окружение. И в нем лежит лишь звенящая пустота. Но благодаря [Environment] counter ссылка будет использована как ссылка на внешнее лексическое окружение, которое даст нам доступ по ссылке к переменам предшествующего вызова, где counter и был создан.



Вызов начнет искать переменную count у себя дома, но когда не сможет её найти, потому что там пусто, он обратится в лексическое окружение предшественника и найдёт её там.   
Сборщик мусора не будет убивать те переменные, которые содержат ссылки на что то, в нашем случае на функцию с [Environment].  
Уничтожение будет происходить лишь после потери всех ссылок на объект, который используется.  
**Шаг 6**

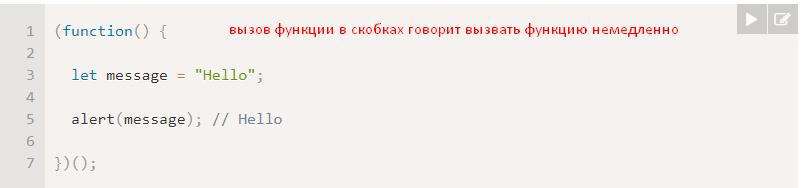
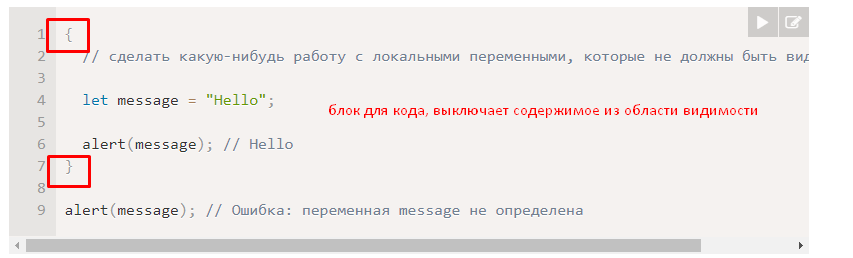


Итого:



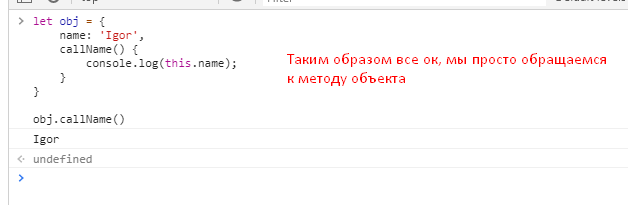
!if, циклы тоже имеют свое лексическое окружение. Но в отличие от обычных функций, к нему нельзя никак обратиться, оно существует только внутри блока с циклом или условием, снаружи их нельзя увидеть.

**Блоки кода**  
**Немножко древности – LIFE(immediately-invoked function expressions)**

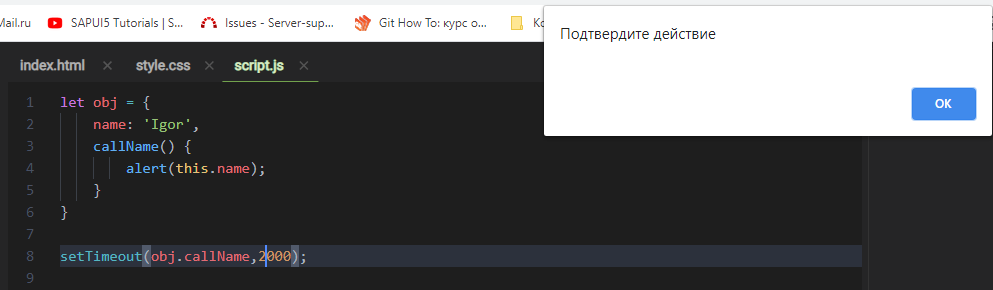


* **Как реализовать функцию bind?**

Эта тема освещалась выше и напрямую связана с замыканиями.   
Разберемся более детально. Если передать метод, отдельно от объекта, то произойдет потеря this.  
Чтобы этого избежать и существует метод bind.



Но если этот метод передать, например, в SetTimeout, то произойдет потеря this



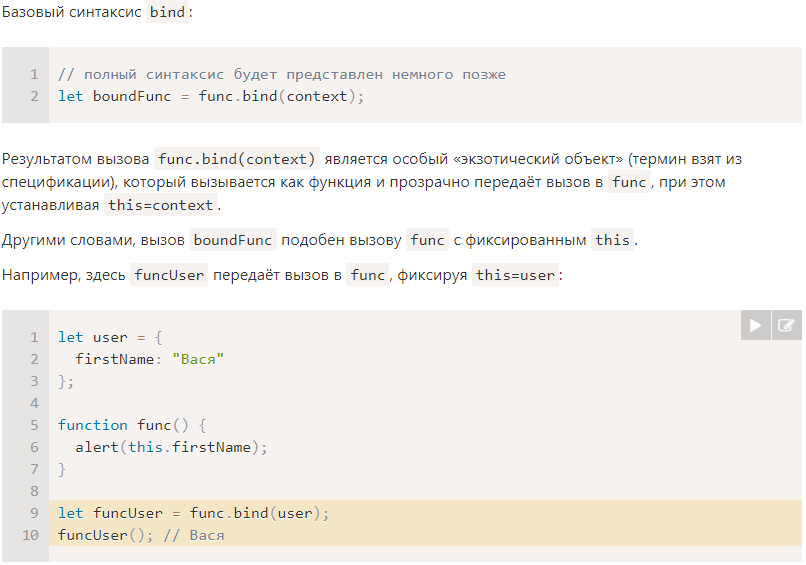
А это произошло потому что у SetTimeout есть особенность, он устанавливает this глобального объекта, то есть window. В нашем случае, это равносильно такому обращению window.callName, естественно, вернется undefined.  
Здесь можно реализовать два решения этой проблемы:

**Решение 1**



**Решение 2**

И им является самый надежный способ – вызвать метод bind()



* **Что такое callback(функция обратного вызова)? Когда обычно применяется?**

Многие действия в JS асинхронные:

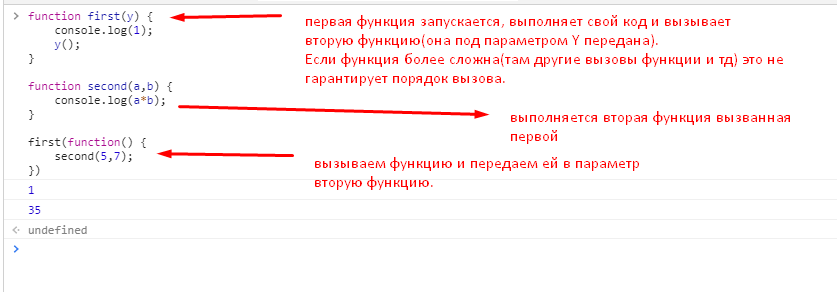


В этом и принцип асинхронности – выполнить что то потом, а не сейчас, не дожидаться выполнения скрипта, а выполнять другой участок кода.



В этом и есть смысл, передать вторым аргументом функцию(обычно анонимную), которая будет выполняться по завершению действия.  
Теперь глубже:  
Код выполняется движком, функции сразу инитятся. Когда дело доходит до вызова функции – она сразу вызывается, движок не ждет, когда она полностью отработает и вызыввает сразу следующую. Если функция маленькая, она легко сможет отработать и всё будет ок. Но если код большой, либо идёт работа с сервером, это может привести к различным ошибкам.

По своей природе каллбек это функция переданная в качестве параметра.



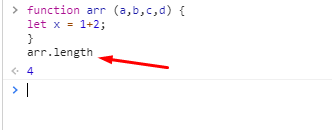
* **Что такое каррирование?**

Каррирование это техника для работы с функциями.

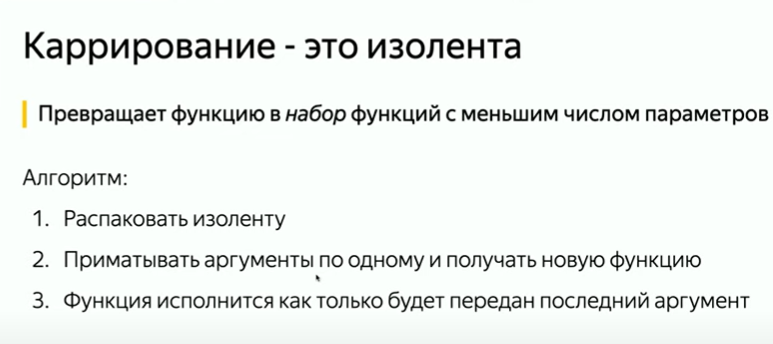
Относиться к функциональному программированию. Разберемся глубже в самих функциях.  
**Параметры функции** – переменные в объявлении функции.

**Аргументы** – конкретные значерия, переданные при вызове функции.

У функции всегда есть длина, и она равна длинне аргументов



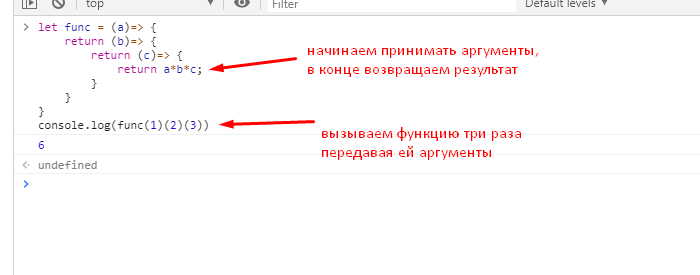
Каррирование как изолента, с помощью которой мы можем примотать аргументы одной функции к другой.



Каррирование всегда возвращает новую функцию.

**Каррирование – это трансформация функций таким образом, чтобы они принимали аргументы не как f(a, b, c), а как f(a)(b)(c), то есть функция принимает не набор аргументов, а всего лишь один.**

Так же можно сказать, что каррирование это процесс превращения функции с несколькими аргументами, в функцию с меньшей арностью. **Арность** – это колличество аргументов функции(например, двухарная функция – функция с двумя аргументами).



* **Что такое частичное применение?**

Частичное применение очень похоже на карирование, но у него другая концепция.  
При частичном применении функцию преобразуют в другую функцию, обладающим меньшим числом аргументов(меньшей арностью).

