СОДЕРЖАНИЕ

[1. Лексическое окружение 3](#_Toc159353501)

[2. Замыкания 4](#_Toc159353502)

[3. Разбор первой задачи. 5](#_Toc159353503)

[4. Область видимости 7](#_Toc159353504)

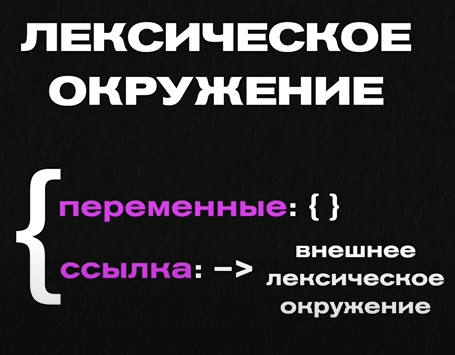
[5. Какие недостатки имеет VAR 9](#_Toc159353505)

# Лексическое окружение

Лексическое окружение — это некоторый невидимый (скрытый) объект, который есть у любого блока, скрипта или функции в JavaScript.

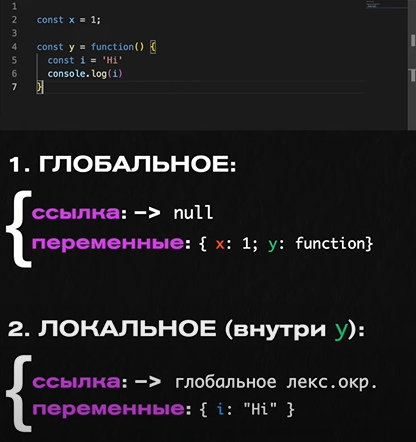
Данный объект состоит из двух частей:

* Объект с переменными текущей области видимости,
* Ссылка на внешнее родительское лексическое окружение.



1. Лексическое окружение.

Разберем примеры:



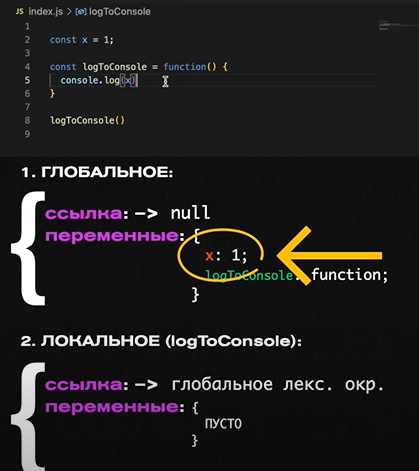
1. Пример лексического окружения.

1 глобальное имеет доступ только к своим переменным.

2 локальное имеет доступ ко всем переменным. Будет создано только во время вызова ф-ции.

# Замыкания

Замыкания — это способность ф-ции в JS запоминать лексическое окружение, в котором она была создана. Т.е. хранить в себе ссылку на это окружение.



1. Замыкания.

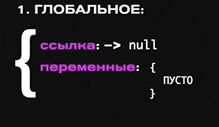
Происходит поиск переменной x, тк в текущем лексическом окружении ее нет, ф-ция смотрит на родительское окружение, это и есть замыкание.

# Разбор первой задачи.



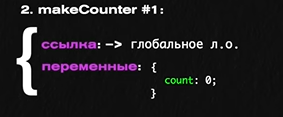
1. Разбор задачи

И так, как будет выполняться этот код и как будут созданы окружения? Сначала создает глобальное лексическое окружение:



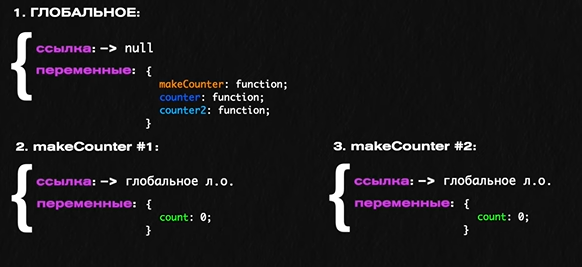
Далее в него добавляется makeCounter: function и происходит первый вызов этой функции, в качестве аргумента передаем 0.

Так вот при вызове этой функции, мы попадаем внутрь функции makeCounter, на этом этапе создается новое лексическое окружение для этой функции и в него добавляется переменная count: 0:



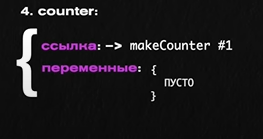
Результат сохраняется в переменной counter и попадает в глобальное лексическое окружение, то же самое со второй.

И так сейчас мы имеет такую картину:



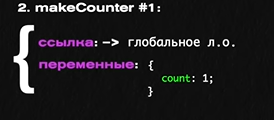
Лексическое окружение для первого вызова и для второго вызова это совершенно разные независимые объекты.

Далее вызываем функцию counter(), что происходит в этот момент? Мы попадаем внутрь функции, которую вернули до этого, то есть на строку 5. Для этого вызова функции также создается свое лексическое окружение:



Но в нем нет лексического окружения, оно пустое, чтобы выполнить count++ мы пытаемся найти переменную и не находим ее в нашем лексическом окружении, поэтому мы обращаемся к родительскому окружению и там находим count, которая равна 0.

Далее мы возвращаем ее и только ПОСЛЕ return прибавляем один. Поэтому первый лог выведет ноль:



Теперь там count уже равен 1. Далее происходит еще один вызов, и происходит все то же самое, не находим в новом лексическом окружении count, идем выше, находим его там и там оно равно уже 1, мы его возвращаем и только потом прибавляем один.

# Область видимости

Область видимости — это зона, в которой переменные доступны, то есть в которой мы к ним переменным можем обратиться.

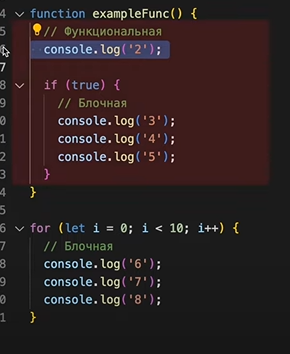
* Глобальная - log находится в глобальной области видимости:



1. Глобальная.

Это самая внешняя область, когда мы создаем просто в js файле так сказать.

* Функциональная — это обл. внутри функции.



1. Функциональная.

* Блочная — это обл внутри любого блока с {} скобками.



1. Блочная.

Внутри области видимости, мы можем видеть переменные из этой области + переменные из родительской.

Наоборот, это не работает, у родительских областей нет доступа к дочерним.

# Какие недостатки имеет VAR

Основная проблема - объявленная переменная таким образом переменная, не ограничивается блоком кода, в котором она была создана. У нее функциональная область видимости. Ее видимость может ограничить только функция. Инструкции типа if while for ее область не ограничивает.

Вторая проблема - такую переменную можно переопределить и пере объявлять.

Третья проблема - такая переменная может быть использована до объявления.

Четвертая проблема - такая переменная создается как св-во для window, поэтому тот же alert можно переопределить.

См js файл!!!