**Что такое замыкание?**

Замыкание – это способность всех функции в JS запоминать лексическое окружение, в котором они были созданы, т.е. хранить в себе ссылку на это окружение.

Все функции помнят лексическое окружение, в котором они были созданы, т.к. они имеют скрытое свойство [[Environment]], которое хранит ссылку на лексическое окружение, в котором они были созданы.

Функция запоминает, где она была создана, независимо от того, где она вызывается.

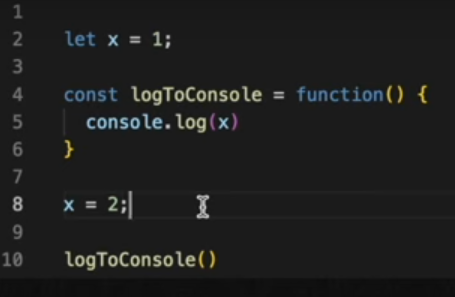
Если функция изменяет переменные, они обновляются в том лексическом окружении, в котором они существуют.

В некоторых языках замыкания невозможны, или функция должна быть написана специальным образом, чтобы получилось замыкание. Но, в JavaScript, все функции изначально являются замыканиями (есть только одно исключение, синтаксис "new Function").

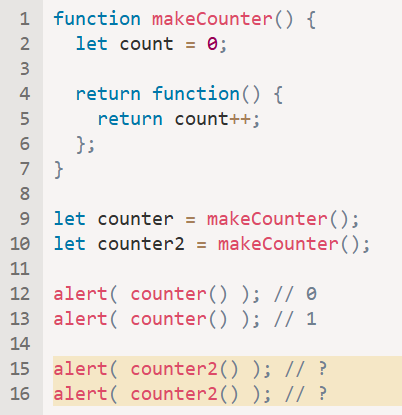
Вложенная функция может быть возвращена: либо в качестве свойства нового объекта (если внешняя функция создаёт объект с методами), либо сама по себе. И затем может быть использована в любом месте. Не важно, где она будет вызвана, она всё так же будет иметь доступ к тем же внешним переменным.

**Задачи.**

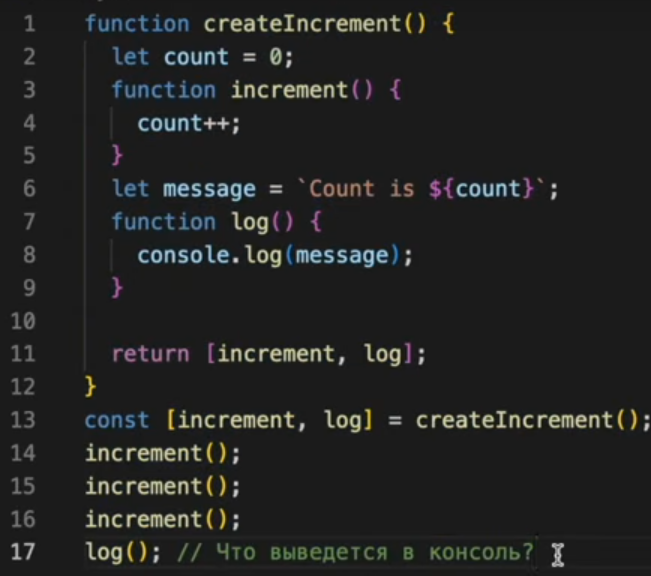
**1. Что будет в консоли? Поч?**



**2. Что будет в алертах? Поч?**



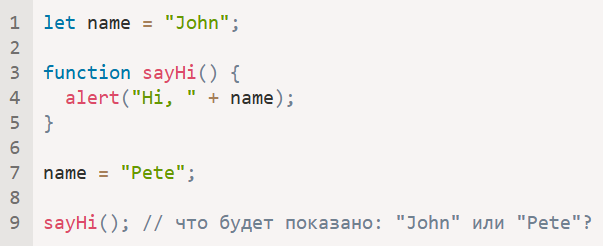
**3. Что выведется в консоль в строке 17?**



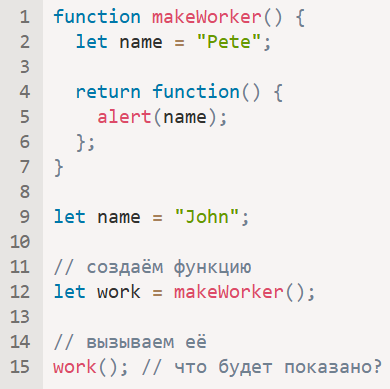
**4. Функция sayHi использует имя внешней переменной. Какое значение будет использоваться при выполнении функции?**

Такие ситуации встречаются как при разработке для браузера, так и для сервера. Функция может быть назначена на выполнение позже, чем она была создана, например, после действия пользователя или сетевого запроса.

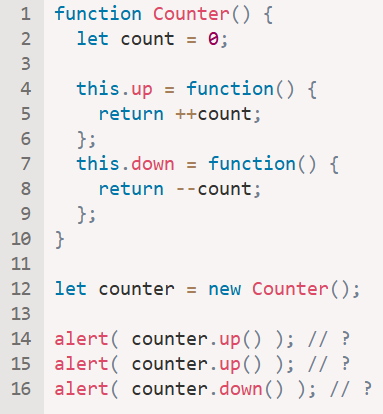
Итак, вопрос: учитывает ли она последние изменения?



**5. Какое значение будет показано? «Pete» или «John»?**



**6. Будет ли работать счетчик? Почему? Что покажет?**



**7. Фильтрация с помощью функции**

Есть встроенный метод arr.filter(f) для массивов. Он фильтрует все элементы с помощью функции f. Если она возвращает true, то элемент добавится в возвращаемый массив.

Сделайте набор «готовых к употреблению» фильтров:

inBetween(a, b) – между a и b (включительно).

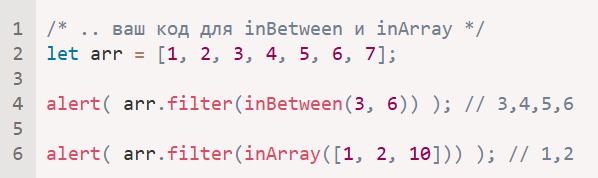
inArray([...]) – находится в данном массиве.

Они должны использоваться таким образом:

arr.filter(inBetween(3,6)) – выбирает только значения между 3 и 6 (включительно).

arr.filter(inArray([1,2,3])) – выбирает только элементы, совпадающие с одним из элементов массива

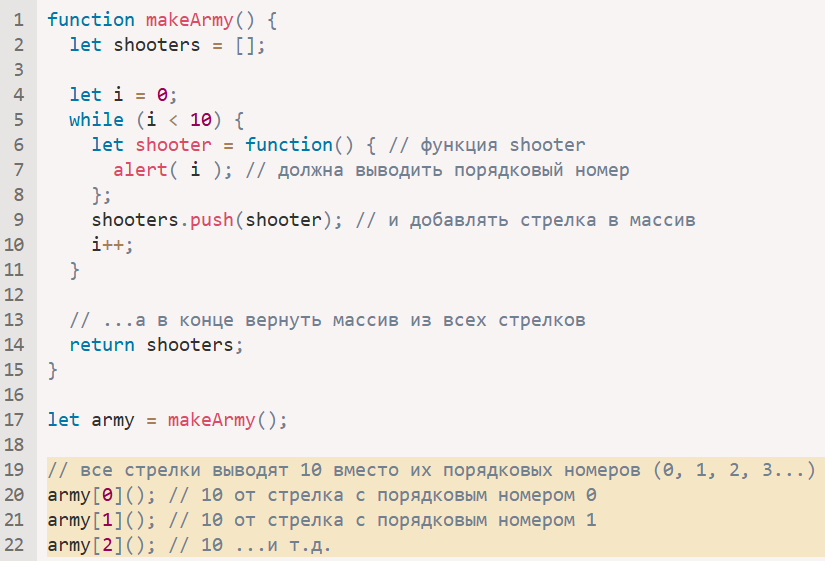
Например:



**8. Почему у всех стрелков одинаковые номера?**

Следующий код создаёт массив из стрелков (shooters).

Каждая функция предназначена выводить их порядковые номера. Но что-то пошло не так…



**Решения.**

1. 2 – т.к. на момент вызова функции x = 2. Берется самое актуальное значение на момент вызова функции.

2. 1й алерт – 0.

2й алерт – 1

3й алерт – 0.

4й алерт – 1

Потому что создается два независимых лексических окружения. Так что у них независимые внешние лексические окружения, у каждого из которых свой собственный count.



3. Будет 0, т.к. переменная message создалась один раз, когда count был равен 0.

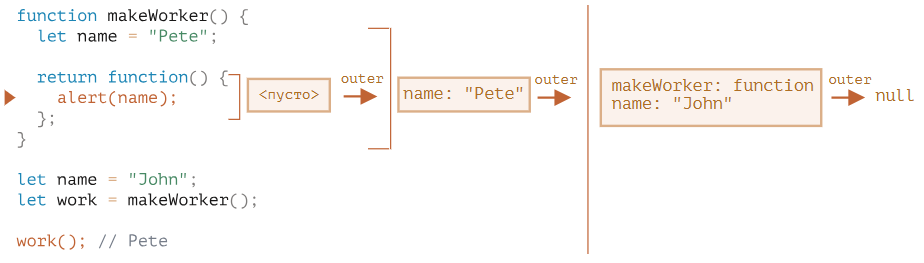
4. Pete.

Функция получает внешние переменные в том виде, в котором они находятся сейчас, она использует самые последние значения.

Старые значения переменных нигде не сохраняются. Когда функция обращается к переменной, она берет текущее значение из своего или внешнего лексического окружения.

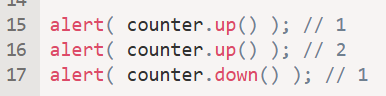
5. Pete.

Функция work() получает name из места его происхождения через ссылку на внешнее лексическое окружение:

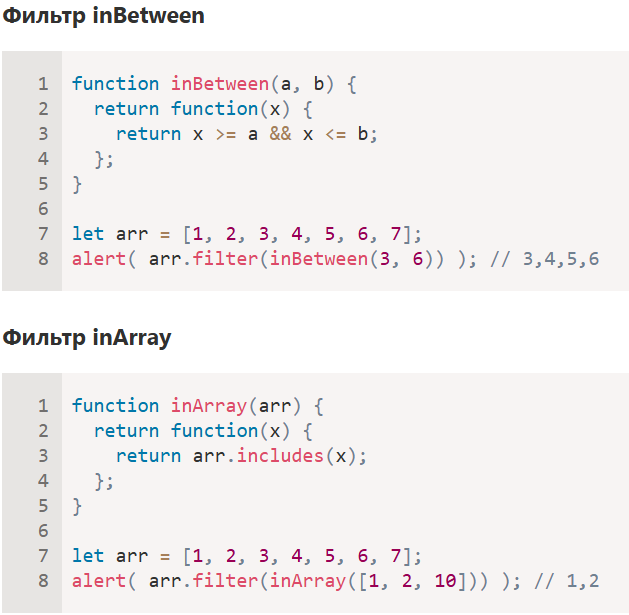


Но если бы в makeWorker() не было let name, то поиск шел бы снаружи и брал глобальную переменную, что мы видим из приведенной выше цепочки. В этом случае результатом было бы "John".

6. Будет работать, т.к. обе вложенные функции были созданы с одним и тем же внешним лексическим окружением, так что они имеют доступ к одной и той же переменной count:



7.



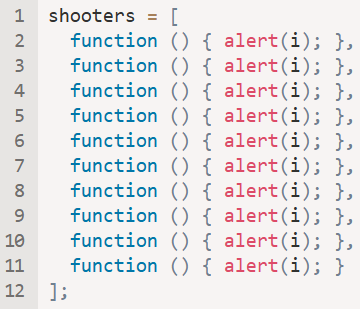
8. Давайте посмотрим, что происходит внутри makeArmy, и решение станет очевидным.

1. Она создаёт пустой массив shooters:



2. В цикле заполняет его shooters.push(function...).

Каждый элемент – это функция, так что получится такой массив:



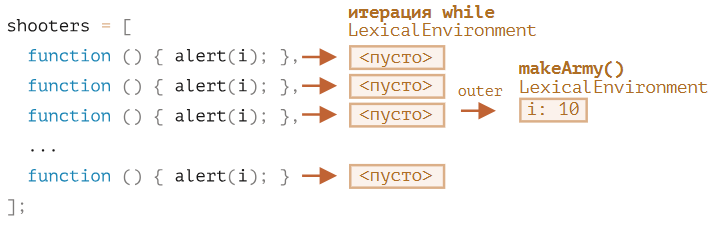
3. Функция возвращает массив.

Позже вызов army[5]() получит элемент army[5] из массива (это будет функция) и вызовет её.

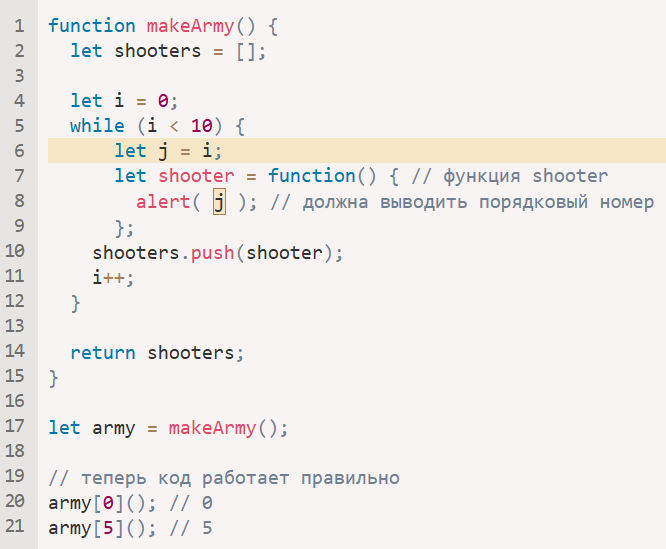
Все эти функции показывают одно и то же потому, что внутри функций shooter нет локальной переменной i. Когда вызывается такая функция, она берёт i из своего внешнего лексического окружения.

Какое будет значение у i? Если посмотреть в исходный код, то мы увидим, что i живёт в лексическом окружении, связанном с текущим вызовом makeArmy(). Но, когда вызывается army[5](), makeArmy уже завершила свою работу, и последнее значение i: 10 (конец цикла while).

Как результат, все функции shooter получат одно и то же значение из внешнего лексического окружения: последнее значение i=10.

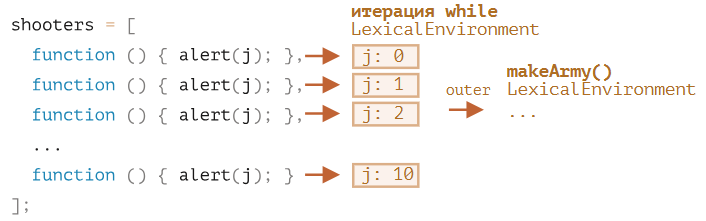


На каждой итерации блока while {...} создается новое лексическое окружение. Чтобы исправить это, можно скопировать значение i в переменную внутри блока while {...}:

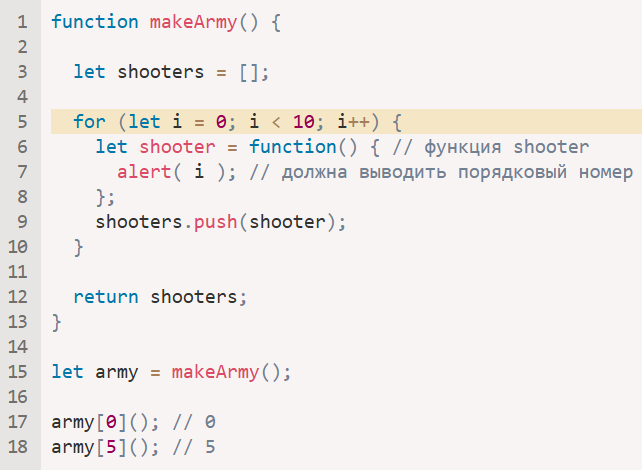


Здесь let j = i объявляет «итерационно-локальную» переменную j и копирует в нее i. Примитивы копируются «по значению», поэтому фактически мы получаем независимую копию i, принадлежащую текущей итерации цикла.

Функции shooter работают правильно, потому что значение i теперь живет чуть ближе. Не в лексическом окружении makeArmy(), а в лексическом окружении, соответствующем текущей итерации цикла:



Этой проблемы также можно было бы избежать, если использовать for:



По сути, это то же самое, поскольку for на каждой итерации создает новое лексическое окружение со своей переменной i. Поэтому функция shooter, создаваемая на каждой итерации, ссылается на свою собственную переменную i, причем именно с этой итерации.

