**Что такое замыкание?**

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

Функции 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.

**В каких случаях замыкания могут быть полезны?**

Вообще, замыкания полезны тем, что позволяют связать данные с функцией, которая работает с этими данными. Данные будут приватными, т.е. их нельзя будет изменить извне, т.е. при помощи замыканий можно организовывать инкапсуляцию данных. Это помогает обеспечить безопасность данных.

Очевидна параллель с объектно-ориентированным программированием, где объекты позволяют связать некоторые данные (свойства объекта) с одним или несколькими методами.

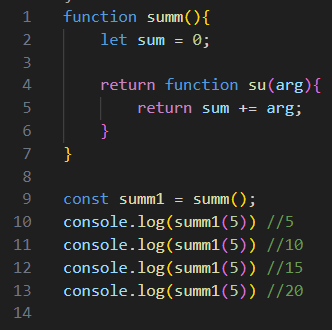
Следовательно, замыкания можно использовать везде, где вы обычно использовали объект с одним единственным методом.

Замыкания могут быть полезны для коллбеков при обработке событий, в фабричных функциях, в функциях с состоянием.

Замыкания используются при каррировании, частичном применении, мемоизации, в декораторах.

**Как сохранить состояние с помощью замыкания, и для чего это состояние может быть использовано?**

Замыкания позволяют сохранять состояния между вызовами функций. Такие функции называются функциями с сохранением состояния, и их можно реализовать при помощи замыкания. Например, нужна функция, которая принимает последовательные числовые значения и вычисляет их сумму. Функция должна помнить ранее переданные значения между вызовами.



**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… |
| **9. Какие из этих 3 функций получают доступ к переменным внешней области видимости (outer scope)?** | **10. Что будет записано в консоль для следующего фрагмента кода (code snippet):** |

|  |  |
| --- | --- |
| **11. Что будет записано в консоль для следующего фрагмента кода (code snippet):** | **12. Что будет записано в консоль для следующего фрагмента кода (code snippet):** |
| **13. Что будет записано в консоль в следующем фрагменте кода (code snippet):** | **14. Напишите функцию multiply(), которая умножает 2 числа**  Если multiply(num1, numb2) будет вызвана с 2 аргументами, то она должна вернуть умножение 2 аргументов.  Но в случае, если вызывается 1 аргумент const anotherFunc = multiply(numb1), то функция должна возвращать другую функцию. Возвращаемая функция при вызове anotherFunc(num2) выполняет умножение num1 \* num2. |

**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.

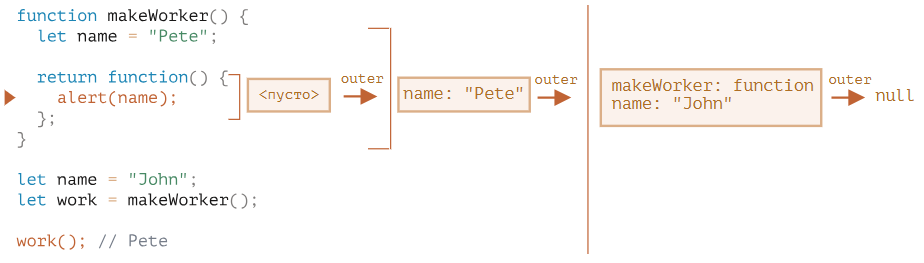
Функция получает внешние переменные в том виде, в котором они находятся сейчас, она использует самые последние значения.

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

5.

Pete.

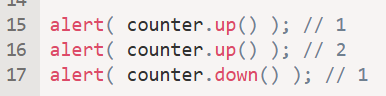
Функция work() получает name из места его происхождения через ссылку на внешнее лексическое окружение:



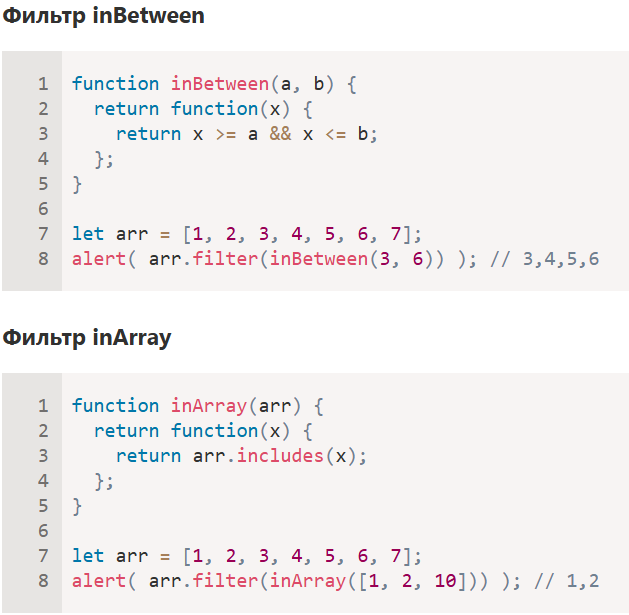
Но если бы в makeWorker() не было let name, то поиск шел бы снаружи и брал глобальную переменную, что мы видим из приведенной выше цепочки. В этом случае результатом было бы "John".

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.

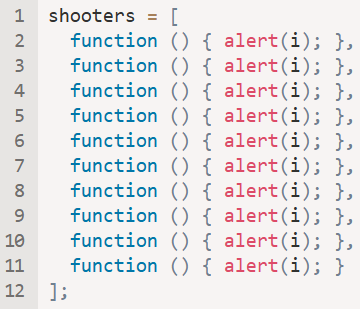
Давайте посмотрим, что происходит внутри makeArmy, и решение станет очевидным.

1. Она создаёт пустой массив shooters:



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

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



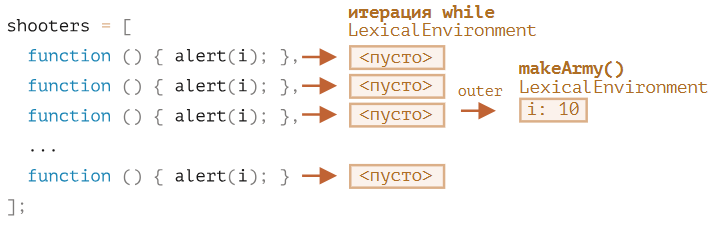
3. Функция возвращает массив.

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

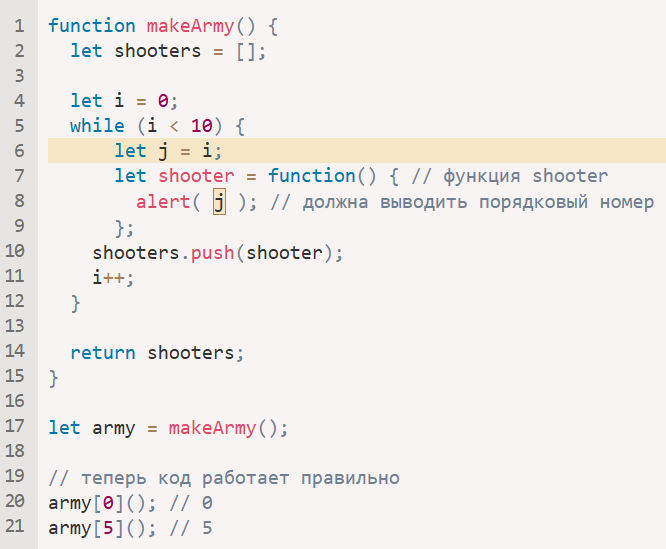
Все эти функции показывают одно и то же потому, что внутри функций shooter нет локальной переменной i. Когда вызывается такая функция, она берёт i из своего внешнего лексического окружения.

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.

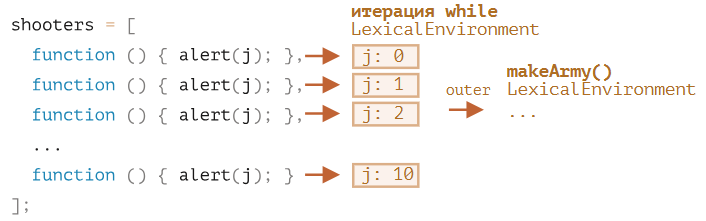


На каждой итерации блока while {...} создается новое лексическое окружение. Чтобы исправить это, можно скопировать значение i в переменную внутри блока while {...}:

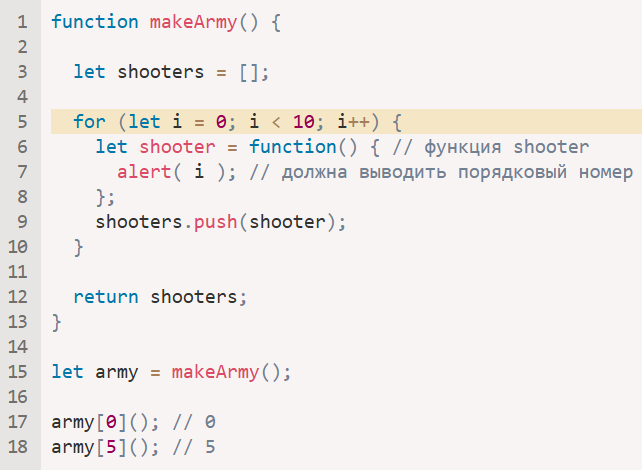


Здесь let j = i объявляет «итерационно-локальную» переменную j и копирует в нее i. Примитивы копируются «по значению», поэтому фактически мы получаем независимую копию i, принадлежащую текущей итерации цикла.

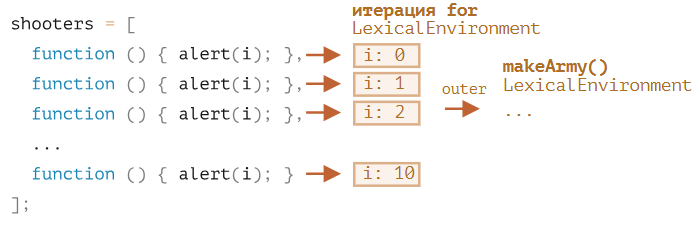
Функции shooter работают правильно, потому что значение i теперь живет чуть ближе. Не в лексическом окружении makeArmy(), а в лексическом окружении, соответствующем текущей итерации цикла:



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.



9.

clickHandler обращается к переменной countClicks из внешней области видимости.

immediate не обращается ни к каким переменным из внешней области видимости.

delayReload обращается к глобальной переменной location из глобальной области видимости (так же известной как крайняя область видимости (outermost scope)).

10.

0 записывается на консоль.

immediateA вызывается с аргументом 0, таким образом, параметр равен 0.

Функция immediateB, будучи вложенной в функцию immediateA, является замыканием, которое захватывает переменную a из внешней области видимости immediateA, где a равно 0. Таким образом, console.log(a) выведет 0.

11.

1 и 0 записываются в консоль.

Первое утверждение let count = 0 объявляет переменную count.

immediate() — это замыкание, которое захватывает переменную count из внешней области видимости. Внутри области видимости функции immediate() count равна 0.

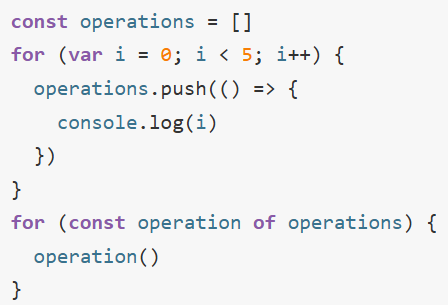
Однако внутри этого условия другая команда let count = 1 объявляет count локальной переменной, которая перезаписывает count из внешней области видимости. Первая console.log(count) записывает 1.

Вторая console.log(count) записывает 0, так как здесь переменная count доступна из внешней области видимости.

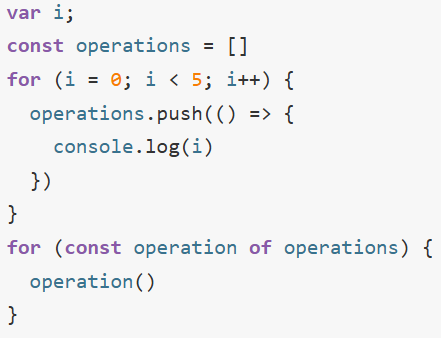
12.

Результатом будет 3, 3, 3.

Код в примере, аналогичен коду ниже – т.е. запрос переменной i происходит после завершения цикла:



Так как объявления переменных var поднимаются в верхнюю часть области видимости, вышеприведённый код аналогичен следующему.



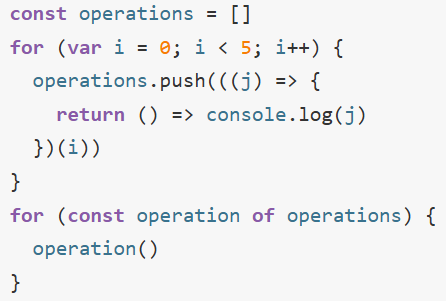
В результате оказывается, что, при запросе i, при вызове operation(), i всё ещё видна, и она уже равна 5, в результате, ссылаясь на i во всех функциях, мы выводим число 5.

Как изменить поведение программы таким образом, чтобы она делала бы то, что от неё ожидается?

Самое простое решение этой проблемы заключается в использовании ключевого слова let. Теперь на каждой итерации цикла каждая функция, добавленная в массив operations, получает собственную копию i, из собственного лексического окружения, создаваемого на каждой итерации.

Ещё один способ решения этой проблемы, который часто применялся до появления стандарта ES6, когда ключевого слова let ещё не было, заключается в использовании IIFE.

При таком подходе значение i сохраняется в замыкании, а в массив попадает функция, возвращаемая IIFE и имеющая доступ к замыканию.

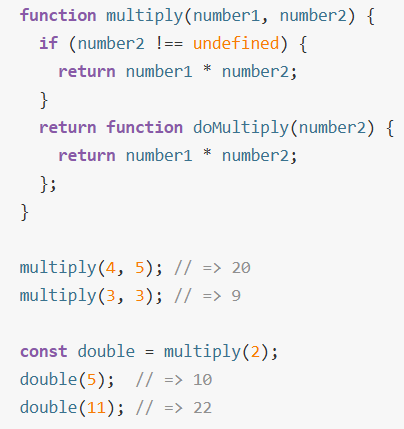


13.

'Count is 0' записывается в консоль.

Переменная message существует в рамках функции createIncrement(), она создалась один раз при вызове функции createIncrement(). Ее начальное значение 'Count is 0', т.к. count на момент создание этой переменной был равен 0. Далее Функция increment() вызывалась 3 раза, в итоге увеличивая count до значения 3, но значение message больше не перезаписывалось и не изменялось.

14.



doMultiply() является замыкающей, поскольку она захватывает переменную number1 из области видимости функции multiply().