**5 - 1 – 1**

1. **Что выведет alert?**
   * Первый **alert**: 1
   * Второй **alert**: 2
   * Третий **alert**: 3
   * Четвертый **alert**: 1
2. **Почему так?**
   * Функция **makeCounter** возвращает другую функцию, которая использует переменную **currentCount** из своего лексического окружения. Каждый вызов **makeCounter** создает новый экземпляр лексического окружения с собственной копией **currentCount**.
3. **На что ссылается [[Environment]] функций?**
   * [[Environment]] ссылается на лексическое окружение, в котором была создана функция. В данном случае, каждая созданная функция (возвращаемая **makeCounter**) сохраняет ссылку на своё лексическое окружение, включая переменную **currentCount**.
4. **Что будет содержать LexicalEnvironment при запуске функций?**
   * Когда функция запускается, её лексическое окружение будет содержать переменные и их значения. В данном случае, для каждого вызова **makeCounter**, лексическое окружение будет содержать **currentCount**.
5. **Что хранится в counter?**
   * Переменная **counter** содержит функцию, возвращаемую **makeCounter**, которая имеет доступ к своему лексическому окружению, включая **currentCount**. Каждый вызов **counter()** увеличивает **currentCount** на 1 и возвращает предыдущее значение.
6. **Когда будет вызвана функция (\*)?**
   * Функция **(\*)** будет вызвана каждый раз при вызове **counter()**. В данном коде она вызывается три раза: **alert(counter())**. Каждый вызов увеличивает **currentCount** и возвращает предыдущее значение.

Таким образом, последовательность **alert** будет выводить значения 1, 2, 3, затем создается новый счетчик **counter2**, и его первый вызов возвращает 1, так как у него свое собственное лексическое окружение.

Начало формы

***Глобальный объект.***

Глобальный объект предоставляет переменные и функции, доступные в любом месте программы. По умолчанию это те, что встроены в язык или среду исполнения.

В браузере глобальные функции и переменные, объявленные с помощью var (не let/const!), становятся свойствами глобального объекта:

***Замыкания, внутренняя работа функции, лексическое окружение.***

[Замыкание](https://en.wikipedia.org/wiki/Closure_(computer_science)) – это функция вместе со всеми внешними переменными, которые ей доступны.

В каждой функции в JavaScript создается свое собственное лексическое окружение в момент ее объявления. Это окружение состоит из двух основных компонентов:

1. **Environment Record (запись окружения):** Это место, где происходит хранение переменных и их значений. В зависимости от контекста, это может быть объект или другая структура данных, способная хранить и организовывать информацию о переменных.
2. **Ссылка на внешнее лексическое окружение (Outer Lexical Environment Reference):** Это ссылка на лексическое окружение, в котором данная функция была объявлена. Это обеспечивает связь с внешним контекстом и позволяет функции обращаться к переменным из внешнего контекста, создавая таким образом замыкания (closures).

***Каррирование.***

[Каррирование](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%80%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) – продвинутая техника для работы с функциями. Она используется не только в JavaScript, но и в других языках.

Каррирование – это трансформация функций таким образом, чтобы они принимали аргументы не как f(a, b, c), а как f(a)(b)(c).

Каррирование не вызывает функцию. Оно просто трансформирует её.

Давайте сначала посмотрим на пример, чтобы лучше понять, о чём речь, а потом на практическое применение каррирования.

Создадим вспомогательную функцию curry(f), которая выполняет каррирование функции f с двумя аргументами. Другими словами, curry(f) для функции f(a, b) трансформирует её в f(a)(b).

function curry(f) { // curry(f) выполняет каррирование

return function(a) {

return function(b) {

return f(a, b);

};

};

}

// использование

function sum(a, b) {

return a + b;

}

let curriedSum = curry(sum);

alert( curriedSum(1)(2) ); // 3

***Генераторы.***

Обычные функции возвращают только одно-единственное значение (или ничего).

Генераторы могут порождать (yield) множество значений одно за другим, по мере необходимости. Генераторы отлично работают с перебираемыми объектами и позволяют легко создавать потоки данных.

Основным методом генератора является next(). При вызове он запускает выполнение кода до ближайшей инструкции yield <значение> (значение может отсутствовать, в этом случае оно предполагается равным undefined). По достижении yield выполнение функции приостанавливается, а соответствующее значение – возвращается во внешний код:

Результатом метода next() всегда является объект с двумя свойствами:

* value: значение из yield.
* done: true, если выполнение функции завершено, иначе false.