Частное учреждение образования

«Колледж бизнеса и права»

|  |
| --- |
| УТВЕРЖДАЮ  Заведующий  методическим кабинетом  \_\_\_\_\_\_\_\_\_\_Е.В.Паскал  «\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_ |

|  |  |
| --- | --- |
| Специальность:  2-40 01 01 «Программное обеспечение информационных технологий» | Дисциплина: «Скриптовые языки программирования» |

**Лабораторная работа № 32**

**Инструкционно-технологическая карта**

Тема: Замыкание в JavaScript. Лексическое окружение. Вложенные функции. Блоки кода и цикл, IIFE.

Цель: Научиться использовать замыкание, вложенные функции в JavaScript

Время выполнения: 2 часа

# Порядок выполнения работы

1. Изучить теоретические сведения к лабораторной работе.
2. Реализовать алгоритм решения задачи.
3. Отлаженную, работающую программу сдать преподавателю. Работу программы показать с помощью самостоятельно разработанных тестов.
4. Ответить на контрольные вопросы.

# Теоретические сведения

JavaScript – язык с сильным функционально-ориентированным уклоном. Он даёт много свободы. Функция может быть динамически создана, скопирована в другую переменную или передана как аргумент другой функции и позже вызвана из совершенно другого места.

Функция может получить доступ к переменным из внешнего окружения, эта возможность используется очень часто.

## Лексическое Окружение

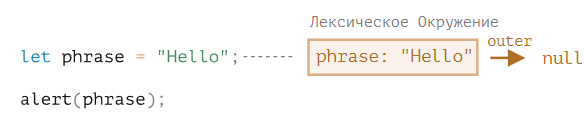
В JavaScript у каждой выполняемой функции, блока кода и скрипта есть связанный с ними внутренний (скрытый) объект, называемый лексическим окружением **LexicalEnvironment**.

Объект лексического окружения состоит из двух частей:

1. Environment **Record** – объект, в котором как свойства хранятся все локальные переменные (а также некоторая другая информация, такая как значение **this**).
2. Ссылка на внешнее лексическое окружение – то есть то, которое соответствует коду снаружи (снаружи от текущих фигурных скобок).

«Переменная» – это просто свойство специального внутреннего объекта: **Environment Record**. «Получить или изменить переменную», означает, «получить или изменить свойство этого объекта».

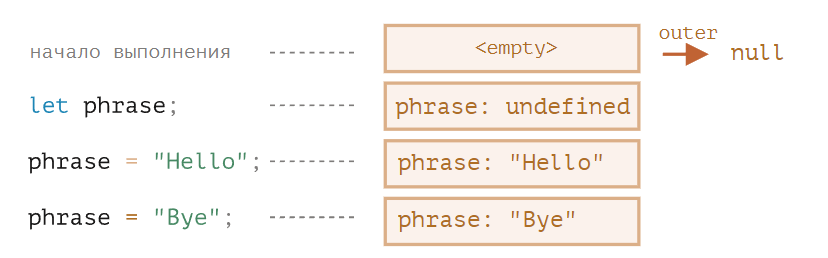
Например, в этом простом коде только одно лексическое окружение:



Это, так называемое, глобальное лексическое окружение, связанное со всем скриптом.

На картинке выше прямоугольник означает **Environment Record** (хранилище переменных), а стрелка означает ссылку на внешнее окружение. У глобального лексического окружения нет внешнего окружения, так что она указывает на **null**.

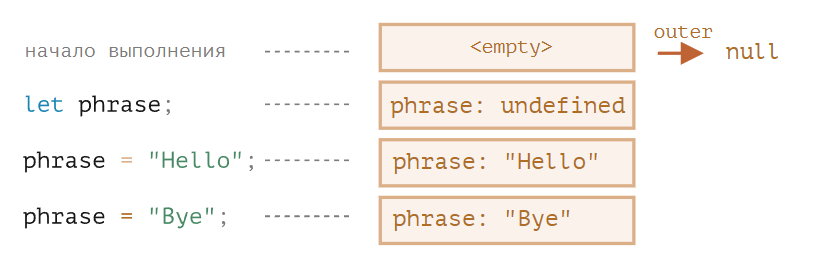
Вот как оно изменяется при объявлении и присваивании переменной:



Это, так называемое, глобальное лексическое окружение, связанное со всем скриптом.

На картинке выше прямоугольник означает **Environment Record** (хранилище переменных), а стрелка означает ссылку на внешнее окружение. У глобального лексического окружения нет внешнего окружения, так что она указывает на **null**.

Вот как оно изменяется при объявлении и присваивании переменной:



Прямоугольники с правой стороны демонстрируют, как глобальное лексическое окружение изменяется в процессе выполнения кода:

1. В начале скрипта лексическое окружение пустое.
2. Появляется определение переменной **let phrase**. У неё нет присвоенного значения, поэтому присваивается **undefined**.
3. Переменной **phrase** присваивается значение.
4. Переменная **phrase** меняет значение.

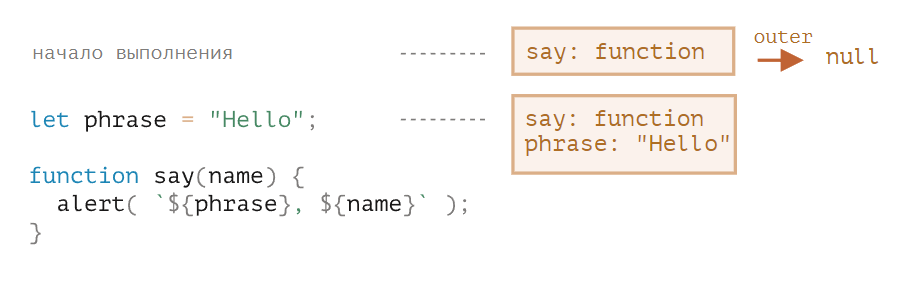
#### Function Declaration.

В отличие от переменных, объявленных с помощью **let**, они полностью инициализируются не тогда, когда выполнение доходит до них, а раньше, когда создаётся лексическое окружение.

Для верхнеуровневых функций это означает момент, когда скрипт начинает выполнение.

Вот почему можно вызвать функцию, объявленную через **Function Declaration**, до того, как она определена.

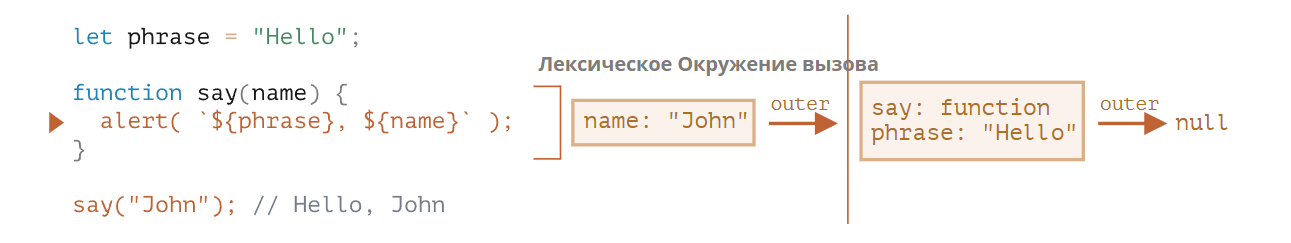
Следующий код демонстрирует, что уже с самого начала в лексическом окружении что-то есть. Там есть **say**, потому что это **Function Declaration**. И позже там появится **phrase**, объявленное через **let**:



#### Внутреннее и внешнее лексическое окружение

При запуске функции для неё автоматически создаётся новое лексическое окружение, для хранения локальных переменных и параметров вызова.

Например, для **say("John")** это выглядит так (выполнение находится на строке, отмеченной стрелкой):



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

1. 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".
2. Внешнее лексическое окружение – это глобальное лексическое окружение. В нём находятся переменная **phrase** и сама функция.

У внутреннего лексического окружения есть ссылка **outer** на внешнее.

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.

Если переменная не была найдена, это будет ошибкой в **strict mode**. Без **strict mode**, для обратной совместимости, присваивание несуществующей переменной создаёт новую глобальную переменную с таким именем.

## Вложенные функции

Функция называется «вложенной», когда она создаётся внутри другой функции.

function sayHiBye(firstName, lastName) {

    // функция-помощник, которую мы используем ниже

    function getFullName() {

        return firstName + " " + lastName;

    }

    alert("Hello, " + getFullName());

    alert("Bye, " + getFullName());

}

Здесь вложенная функция **getFullName()** создана для удобства. Она может получить доступ к внешним переменным и, значит, вывести полное имя. В JavaScript вложенные функции используются очень часто.

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 всё так же будет иметь доступ к тем же внешним переменным.

Здесь создаётся и возвращается функция «счётчик»:

function makeCounter() {

    let count = 0;

    return function () {

        return count++; // есть доступ к внешней переменной "count"

    };

}

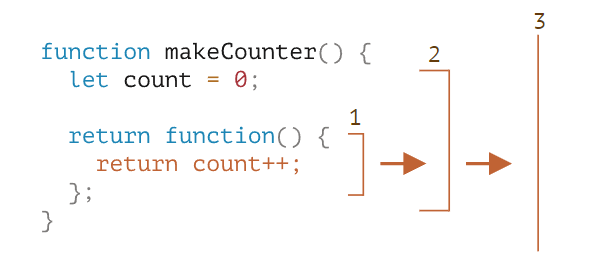
let counter = makeCounter();

alert(counter()); // 0

alert(counter()); // 1

alert(counter()); // 2

Когда внутренняя функция начинает выполняться, начинается поиск переменной **count++** изнутри-наружу. Для примера выше порядок будет такой:



1. Локальные переменные вложенной функции…
2. Переменные внешней функции…
3. И так далее, пока не будут достигнуты глобальные переменные.

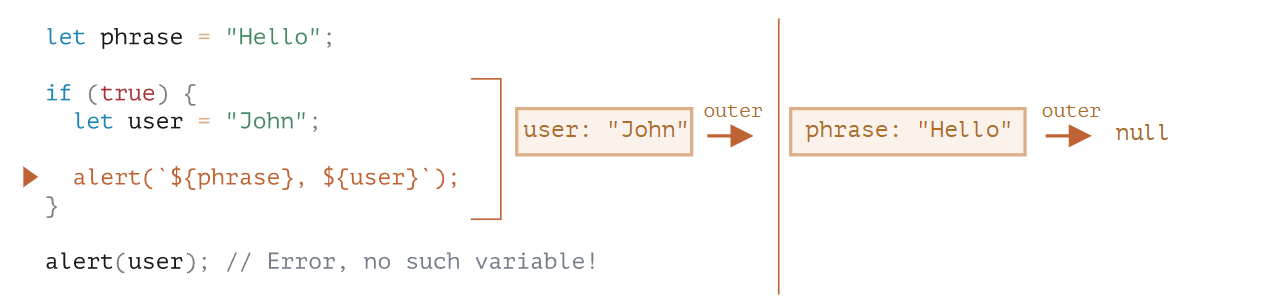
## Блоки кода и циклы, IIFE

Предыдущие примеры сосредоточены на функциях. Но лексическое окружение существует для любых блоков кода {...}.

Лексическое окружение создаётся при выполнении блока кода и содержит локальные переменные для этого блока.

#### If

В следующем примере переменная **user** существует только в блоке **if**:



Когда выполнение попадает в блок **if**, для этого блока создаётся новое лексическое окружение.

У него есть ссылка на внешнее окружение, так что **phrase** может быть найдена. Но все переменные, **Function Expression**, а так же **Function Declaration** в строгом режиме, объявленные внутри **if**, остаются в его лексическом окружении и не видны снаружи.

Например, после завершения **if** следующий **alert** не увидит **user**, что вызовет ошибку.

#### For, while

Для цикла у каждой итерации своё отдельное лексическое окружение. Если переменная объявлена в **for(let ...)**, то она также в нём:

for (let i = 0; i < 10; i++) {

    // У каждой итерации цикла своё собственное лексическое окружение

    // {i: value}

}

alert(i); // Ошибка, нет такой переменной

#### Блоки кода

Можно использовать «простые» блоки кода {...}, чтобы изолировать переменные в «локальной области видимости».

Например, в браузере все скрипты (кроме **type="module"**) разделяют одну общую глобальную область. Так что, если создать глобальную переменную в одном скрипте, она станет доступна и в других. Но это становится источником конфликтов, если два скрипта используют одно и то же имя переменной и перезаписывают друг друга.

Чтобы избежать этого можно использовать блок кода для изоляции всего скрипта или какой-то его части:

{// сделать какую-нибудь работу с локальными переменными, которые не должны быть видны снаружи

    let message = "Hello";

    alert(message); // Hello

}

alert(message); // Ошибка: переменная message не определена

Из-за того, что у блока есть собственное лексическое окружение, код снаружи него (или в другом скрипте) не видит переменные этого блока.

#### IIFE

В прошлом в JavaScript не было лексического окружения на уровне блоков кода.

Так что программистам пришлось что-то придумать. И то, что они сделали, называется «immediately-invoked function expressions» (аббревиатура IIFE), что означает функцию, запускаемую сразу после объявления.

Это не то, что нужно использовать сегодня, но, так как это можно встретить в старых скриптах, полезно понимать принцип работы.

IIFE выглядит так:

(function () {

    let message = "Hello";

    alert(message); // Hello

})();

Здесь создаётся и немедленно вызывается **Function Expression**. Так что код выполняется сразу же и у него есть свои локальные переменные.

**Function Expression** обёрнуто в скобки **(function {...})**, потому что, когда JavaScript встречает **"function"** в основном потоке кода, он воспринимает это как начало **Function Declaration**. Но у **Function Declaration** должно быть имя, так что такой код вызовет ошибку:

// Объявить и сразу же вызвать функцию

function() { // <-- Error: Unexpected token (

    let message = "Hello";

    alert(message); // Hello

} ();

# Контрольные вопросы

1. Что произойдёт, если переменная не будет найдена во внутреннем лексическом окружении?
2. Какая функция называется вложенной?
3. Что такое блочная видимость и для чего она может применяться?
4. Для чего применяется IIFE?
5. Из чего состоит лексическое окружение?

# Задания

**Задание 1**. Напишите функцию **sum**, которая работает таким образом: **sum(a)(b) = a+b**.

Именно таким образом, используя двойные круглые скобки (не опечатка).

Например:

sum(1)(2) = 3

sum(5)(-1) = 4

**Задание 2**. 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:

let arr = [1, 2, 3, 4, 5, 6, 7];

alert( arr.filter(inBetween(3, 6)) ); // 3,4,5,6

alert( arr.filter(inArray([1, 2, 10])) ); // 1,2

**Задание 3.** Сортировать по полю.

Есть массив объектов, который нужно отсортировать:

let users = [

    { name: "John", age: 20, surname: "Johnson" },

    { name: "Pete", age: 18, surname: "Peterson" },

    { name: "Ann", age: 19, surname: "Hathaway" }

];

Обычный способ был бы таким:

// по имени (Ann, John, Pete)

users.sort((a, b) => a.name > b.name ? 1 : -1);

// по возрасту (Pete, Ann, John)

users.sort((a, b) => a.age > b.age ? 1 : -1);

Сделайте сортировку следующим образом:

users.sort(byField('name'));

users.sort(byField('age'));

# Литература

1. **Диков, А.В.** Клиентские технологии веб программирования: JavaScript и DOM: учебное пособие / А.В. Диков. – СПб: Лань, 2020 – 124 с.
2. **Читанамбри, Кирупа.** Изучаем React / Кирупа Читанамбри – 2-е изд. – М.: Эксмо, 2019. 368 с.
3. **Хавербеке, Марейн.** Выразительный JavaScript. Современное веб-программирование. / Марейн Хавербеке – 3-е изд. – СПб.: Питер, 2019. – 480 с.:

Преподаватель Рогалевич А.В.

Рассмотрено на заседании цикловой

комиссии программного обеспечения

информационных технологий №10

Протокол № \_\_\_ от \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Председатель ЦК\_\_\_\_\_\_\_\_ В.Ю.Михалевич