Контекст исполнения. Области видимости. Замыкание.

Движок JavaScript имплементирует особый механизм, называемый контекстом исполнения (Execution Context), позволяющий отслеживать текущую точку кода в процессе выполнения программы (скрипта). В JS существует три контекста исполнения:

- глобальный – это сам JS-скрипт
- функциональный – это функция
- eval – код внутри функции eval (запрещена к использованию разработчиками, можете забыть о ней)

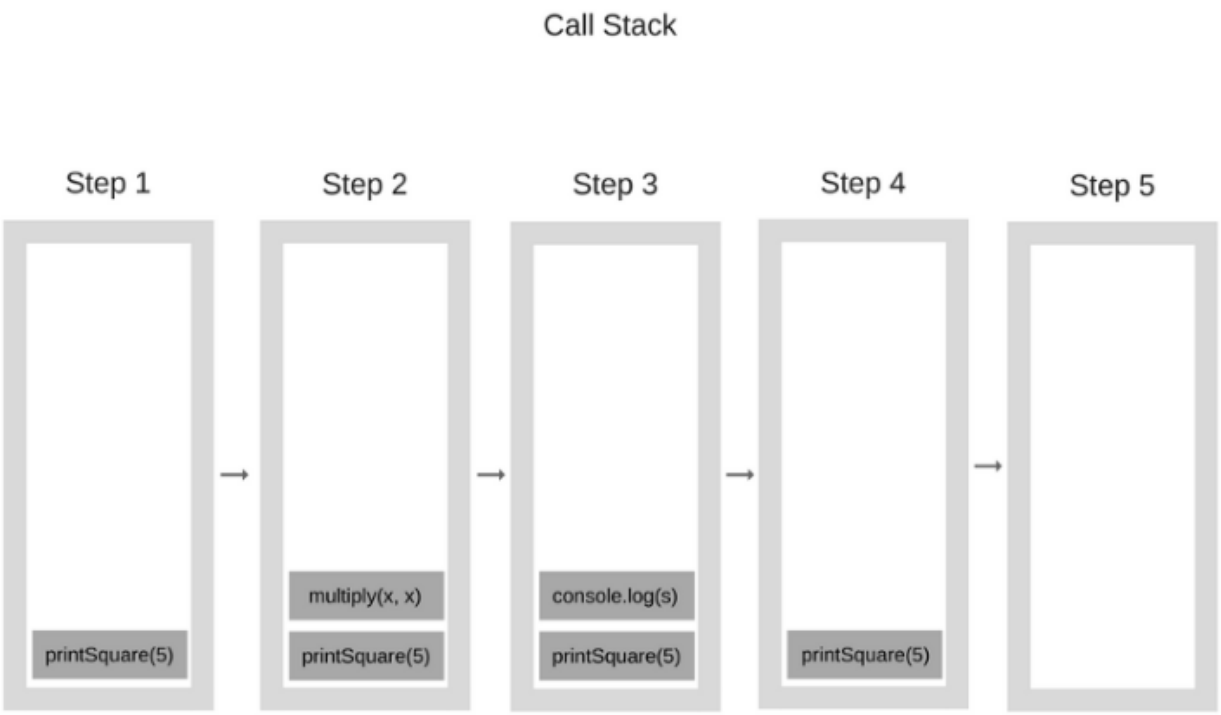
Поскольку JS код однопоточный, то в данный конкретный момент времени процесс может находится только в одном из контекстов исполнения.

Движок также имплементирует стек вызовов (Call Stack) – это структура данных типа стек (LIFO), которая управляет последовательностью контекстов исполнения, поскольку они могут быть вложенными друг в друга. При запуске программы в изначально пустой стек вызовов помещается глобальный контекст исполнения. Далее, если процесс выполнения программы заходит в функцию, в стек помещается контекст исполнения данной функции. Если процесс снова заходит в функцию, то действие повторяется. Иначе, если процесс обработал функцию полностью, то контекст этой функции извлекается из стека вызовов и управление возвращается в ту точку, которая непосредственно следует за точкой вызова только что отработанной функции.

Для наглядности вышесказанного приведём пример, взятый из статьи на Хабр [1] (кстати, рекомендуется Вам её прочесть тоже):

```
function multiply(x, y) {
    return x * y;
}
function printSquare(x) {
    var s = multiply(x, x);
    console.log(s);
}
printSquare(5);
```

Здесь приведён фрагмент простого кода и примерная схема заполнения стека вызовов в процессе выполнения программы с этим кодом:



Вы можете самостоятельно посмотреть на работу стека вызовов в инструментах разработчика в браузере (DevTools) во вкладке «Sources». Для этого нужно поставить точку останова перед вызовом функции. На видео [2] представлена демонстрация такого обзора на примере простого скрипта. Здесь видно, что перед заходом в функцию a() в стеке уже находится некоторая функция с неизвестным именем (anonymous) – в данном случае это глобальный контекст исполнения.

Рассмотренная информация необходима для понимания работы любой программы, написанной на JS. На практике, эти знания могут пригодиться в основном для того, чтобы избежать ошибок переполнения стека вызовов (отведённая под него память ограничена), что особенно актуально при написании рекурсивных функций.

Теперь рассмотрим ещё один механизм, определяющий синтаксис программ – это лексическая область видимости переменных и функций (Lexical Scope). Лексической она называется потому, что определяется во время лексического анализа кода, который осуществляется перед синтаксическим анализом. Для краткости далее будем называть её просто областью видимости.

В JS область видимости – это область кода, заключённая в фигурные скобки «{ … }» или сам файл скрипта. Существуют следующие области видимости:

- глобальная
- модульная
- функциональная
- блочная

Переменные (функции), объявленные в глобальной области видимости (то есть на самом верхнем уровне вложенности) доступны во всём коде скрипта – в каждой функции, блоке и даже модуле. Модуль по смыслу не является самым верхним уровнем вложенности, следовательно, объявить глобальную переменную (функцию) там нельзя:

```
<html>
    <head>
        <script>
            "use strict";

            const global_const = "global_const";

            function global_func() {
                return "global_func";
            }

            console.log(module_const); // Reference Error
            console.log(module_func()); // Reference Error
        </script>
        <script type="module">
            const module_const = "module_const";

            function module_func() {
                return "module_func";
            }

            console.log(global_const); // "global_const"
            console.log(global_func()); // "global_func"
        </script>
    </head>
    <body></body>
</html>
```

Переменные (функции), объявленные на верхнем уровне вложенности модуля, то есть в модульной области видимости, видны в каждой функции и блоке, объявленных в модуле. Необходимо отметить, что возможно объявить псевдоглобальную переменную (функцию) в модуле, если присвоить её как свойство глобального объекта window. Однако, это является плохой практикой. В случае модулей лучше всего использовать директивы import и export. Если Вы пока не знаете, что такое модули, то можете пока пропустить упоминания о них, поскольку впереди им будет посвящена целая тема.

Переменные (функции), объявленные внутри тела функции (функциональная область видимости), доступны внутри тела этой функции и всех вложенных в неё функций. Скорее всего Вы уже сталкивались с этим фактом в процессе изучения языков программирования.

И, наконец, блочная область видимости определяется обрамлением одного или нескольких выражений фигурными скобками. Это например, блок в условном операторе if (условие) { блок 1 } else { блок 2 }; в switch-case, и т. д. Блочная область видимости может быть задана также в любом месте кода с помощью фигурных скобок, например:

```
<!DOCTYPE html>

<html>
    <head>
        <script>
            "use strict";

            {
                const a = 100;
            }

            console.log(a); // ReferenceError
        </script>
    </head>
    <body></body>
</html>
```

Эта возможность появилась с введением ключевых слов «let» и «const» в стандарте EcmaScript 2015. Условия видимости переменных (функций) в блоке те же, что и в функциональной области видимости.

Переменные в JS, объявленные с помощью ключевого слова «var», не распознают блочную область видимости, в то время как «let» и «const» распознают все существующие области видимости. Использование «var» усложняет отладку кода вследствие своих особенностей – переобъявления себя же и «всплытия» (hoising) в пределах глобальной (модульной) и функциональной областях видимости. Поскольку почти все современные браузеры поддерживают стандарт EcmaScript 2015, использование «var» является плохой практикой. Подробнее о переменных Вы можете узнать в статьях [3] и [4].

С областями видимости непосредственно связано замыкание (Closure) – это механизм, обеспечивающий доступ к переменным и функциям внутри некоторой функции, которые объявлены (или инициализированы) в области видимости, являющейся внешней к этой функции. Замыкание формируется в процессе объявления функции, а не в процессе её вызова!

Рассмотрим пример из Доки [3]:

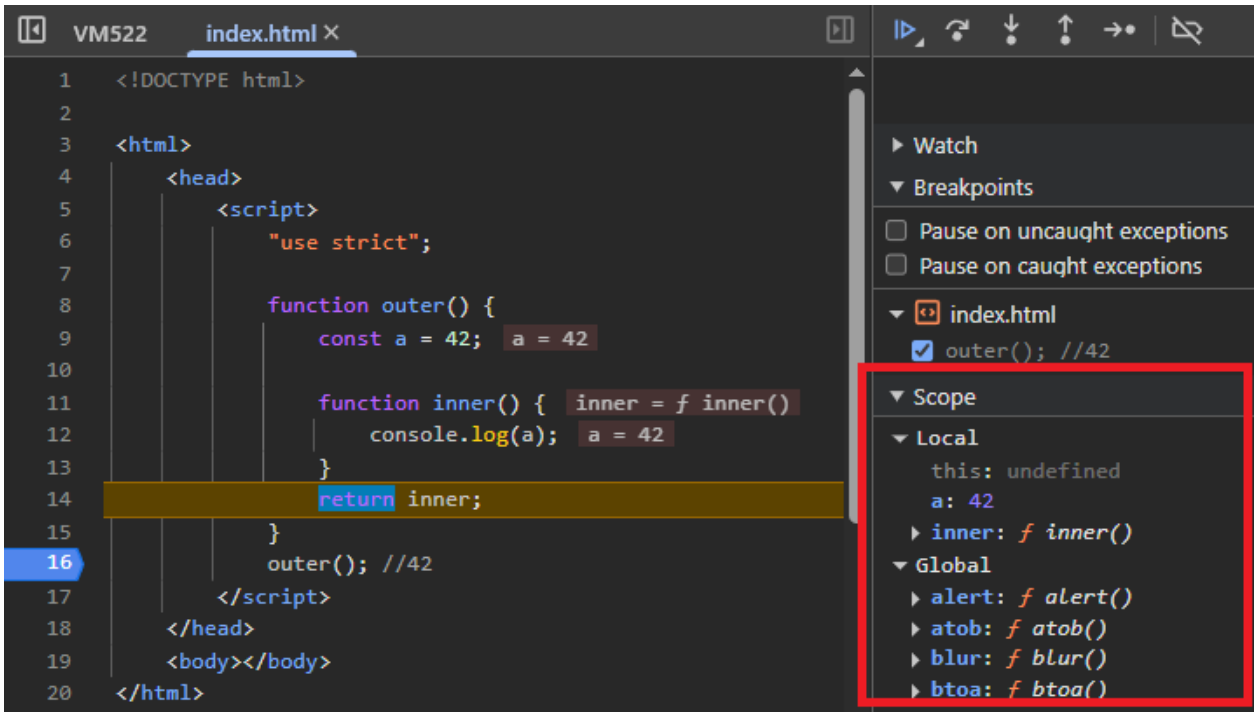
```
<script>
    "use strict";

    function outer() {
        const a = 42;

        function inner() {
            console.log(a);
        }
        return inner;
    }
    outer(); //42
</script>
```

Здесь в процессе объявления функции inner() для неё создается замыкание, включающее в себя внешнюю область видимости, то есть область видимости функции outer(), а также глобальную. В общем

случае замыкание включает в себя все родительские области видимости по отношению к рассматриваемой функции. В свою очередь, для функции outer() тоже создается замыкание, включающее в себя глобальную область видимости.

В момент вызова функции inner(), она будет знать о переменной «a», поскольку имеет доступ к замыканию и использует его. При этом, переменная «a» не видна из областей видимости, внешних по отношению к функции outer(). Рекомендуется прочитать статью [5], чтобы глубже понять принцип работы замыканий и их назначение.

Проанализировать области видимости функций также можно в DevTools:



Вопросы и задания

1. Каким принципом следует руководствоваться при выборе между const и let при необходимости инициализации (объявлении) переменной? Приведите аргументы, почему это важно?

2. Чем отличается объявление переменной от инициализации? В чём заключается отличие инициализации переменной const от let? А переменных const и let от var? В чём заключается отличие объявления const и let от var?

3. Что будет выведено в консоль под номерами 1 и 2 при выполнении скрипта, представленного на скриншоте ниже? Объясните, почему так происходит.

```
<script>
    "use strict";

    let name = "Mouse";
    var surname = "Greys";

    if (true) {
        let name = "Dog";
        var surname = "Brown";

        console.log(1, `${name} ${surname}`); // ??
    }

    console.log(2, `${name} ${surname}`); // ??
</script>
```

4. В чём заключается принципиальное отличие способов объявления функций: Function Declaration от Function Expression? Как оно влияет на поднятие (hoisting)?

5. Чем отличаются стрелочные функции от обычных? Какие преимущества и недостатки Вы можете выделить у стрелочных функций по сравнению с обычными?

6. В следующем коде представлен цикл с использованием переменной var i. Представьте, что Вы в 2000-ом году и переменные const и let ещё не реализованы в стандарте. Вам необходимо каждую секунду выводить в консоль число – переменную цикла i.

```
for (var i = 0; i < 10; i++) {
    setTimeout(() => {
        console.log(i);
    }, 1000 * i);
}
```

Однако приведённый код работает неправильно – каждую секунду он выводит число 10. Как необходимо изменить код внутри тела цикла for, чтобы числа выводились по порядку от 0 до 9 с интервалом 1 секунда? Воспользуйтесь знаниями о замыканиях для решения этой задачи.

Ссылки

[1]. https://habr.com/ru/companies/ruvds/articles/337042/

[2]. https://disk.yandex.ru/i/aamHtTMjTFqESQ

[3]. https://doka.guide/js/closures/

[4]. https://doka.guide/js/var-let/

[5]. https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Closures