\_\_\_\_\_ - понятно полностью

\_\_\_\_\_ - не до конца

\_\_\_\_\_ - ваще не понятно

\_\_\_\_\_ - мои вопросы

Контекст выполнения  
Контекст выполнения (execution context) — это, если говорить упрощённо, концепция, описывающая окружение, в котором производится выполнение кода на JavaScript. Код всегда выполняется внутри некоего контекста.

### Типы контекстов выполнения В JavaScript существует три типа контекстов выполнения:

* Глобальный контекст выполнения. Это базовый, используемый по умолчанию контекст выполнения. Некий код, который находится не внутри какой-нибудь функции, принадлежит глобальному контексту. Глобальный контекст характеризуется наличием глобального объекта, которым, в случае с браузером, является объект window, и тем, что ключевое слово this указывает на этот глобальный объект. В программе может быть лишь один глобальный контекст.
* Контекст выполнения функции. Каждый раз, когда вызывается функция, для неё создаётся новый контекст. Каждая функция имеет собственный контекст выполнения. В программе может одновременно присутствовать множество контекстов выполнения функций. При создании нового контекста выполнения функции он проходит через определённую последовательность шагов.
* Контекст выполнения функции eval. Код, выполняемый внутри функции eval, также имеет собственный контекст выполнения. Однако функцией eval пользуются очень редко, поэтому здесь об этом не говорится.

## ЧТО ТАКОЕ Стек выполнения ? Стек выполнения (execution stack), его ещё называют стеком вызовов (call stack), это LIFO-стек(last in, first out, «последним пришёл — первым ушёл»), который используется для хранения контекстов выполнения, создаваемых в ходе работы кода.

## Интерпретатор JavaScript всегда начинает выполнять код с глобального контекста (в браузере это объект window).

## Когда JS-движок начинает обрабатывать скрипт, движок создаёт глобальный контекст выполнения и помещает его в текущий стек. При обнаружении команды вызова функции движок создаёт новый контекст выполнения для этой функции и помещает его в верхнюю часть стека. Движок выполняет функцию, контекст выполнения которой находится в верхней части стека. Когда работа функции завершается, её контекст извлекается из стека и управление передаётся тому контексту, который находится в предыдущем элементе стека.

## 

## Вот как будет меняться стек вызовов при выполнении этого кода.

## 

Рисунок Состояние стека вызовов

## Когда вышеприведённый код загружается в браузер, JavaScript-движок создаёт глобальный контекст выполнения и помещает его в текущий стек вызовов. При выполнении вызова функции first() движок создаёт для этой функции новый контекст и помещает его в верхнюю часть стека.

## При вызове функции second() из функции first() для этой функции создаётся новый контекст выполнения и так же помещается в стек. После того, как функция second() завершает работу, её контекст извлекается из стека и управление передаётся контексту выполнения, находящемуся в стеке под ним, то есть, контексту функции first().

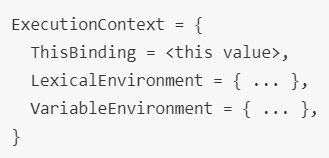
## Когда функция first() завершает работу, её контекст извлекается из стека и управление передаётся глобальному контексту. После того, как весь код оказывается выполненным, движок извлекает глобальный контекст выполнения из текущего стека.

## О создании контекстов и о выполнении кода До сих пор мы говорили о том, как JS-движок управляет контекстами выполнения. Теперь поговорим о том, как контексты выполнения создаются, и о том, что с ними происходит после создания. В частности, речь идёт о стадии создания контекста выполнения и о стадии выполнения кода.

### Стадия создания контекста выполнения Перед выполнением JavaScript-кода создаётся контекст выполнения. В процессе его создания выполняются три действия:

1. Определяется значение this и осуществляется привязка this (this binding).
2. Создаётся компонент LexicalEnvironment (лексическое окружение).
3. Создаётся компонент VariableEnvironment (окружение переменных).

Концептуально контекст выполнения можно представить так:



#### **Привязка this**

#### В глобальном контексте выполнения this содержит ссылку на глобальный объект (как уже было сказано, в браузере это объект window). В контексте выполнения функции значение this зависит от того, как именно была вызвана функция. Если она вызвана в виде метода объекта, тогда значение this привязано к этому объекту. В других случаях this привязывается к глобальному объекту или устанавливается в undefined (в строгом режиме).

#### 

#### Рассмотрим пример:

let foo = {

 baz: function() {

 console.log(this);

 }

}

foo.baz();    // 'this' указывает на объект 'foo', так как функция 'baz' была вызвана

            // как метод объекта 'foo'

let bar = foo.baz;

bar();       // 'this' указывает на глобальный объект window, так как при вызове функции

            // ссылка на объект не используется

#### **Лексическое окружение** В соответствии со [спецификацией](http://ecma-international.org/ecma-262/6.0/#sec-lexical-environments) ES6, лексическое окружение (Lexical Environment) — это термин, который используется для определения связи между идентификаторами и отдельными переменными и функциями на основе структуры лексической вложенности ECMAScript-кода. Лексическое окружение состоит из записи окружения (Environment Record) и ссылки на внешнее лексическое окружение, которая может принимать значение null. Проще говоря, лексическое окружение — это структура, которая хранит сведения о соответствии идентификаторов и переменных. Под «идентификатором» здесь понимается имя переменной или функции, а под «переменной» — ссылка на конкретный объект (в том числе — на функцию) или примитивное значение.

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

В лексическом окружении имеется два компонента:

1. Запись окружения (Environment Record). Это место, где хранятся объявления переменных и функций.
2. Ссылка на внешнее окружение. Наличие такой ссылки говорит о том, что у лексического окружения есть доступ к родительскому лексическому окружению (области видимости).

Существует два типа лексических окружений:

1. Глобальное окружение (или глобальный контекст выполнения) — это лексическое окружение, у которого нет внешнего окружения. Ссылка глобального окружения на внешнее окружение представлена значением null. В глобальном окружении (в записи окружения) доступны встроенные сущности языка (такие, как Object, Array, и так далее), которые связаны с глобальным объектом, там же находятся и глобальные переменные, определённые пользователем. Значение this в этом окружении указывает на глобальный объект.
2. Окружение функции, в котором, в записи окружения, хранятся переменные, объявленные пользователем. Ссылка на внешнее окружение может указывать как на глобальный объект, так и на внешнюю по отношении к рассматриваемой функции функцию.

## Существует два типа записей окружения:

## Декларативная запись окружения, которая хранит переменные, функции и параметры.

## Объектная запись окружения, которая используется для хранения сведений о переменных и функциях в глобальном контексте.

## В результате, в глобальном окружении запись окружения представлена объектной записью окружения, а в окружении функции — декларативной записью окружения.

## Обратите внимание на то, что в окружении функции декларативная запись окружения, кроме того, содержит объект arguments, который хранит соответствия между индексами и значениями аргументов, переданных функции, и сведения о количестве таких аргументов.

## Лексическое окружение можно представить в виде следующего псевдокода:

## 

## Окружение переменных

## Окружение переменных (Variable Environment) — это тоже лексическое окружение, запись окружения которого хранит привязки, созданные посредством команд объявления переменных (VariableStatement) в текущем контексте выполнения.

## Так как окружение переменных также является лексическим окружением, оно обладает всеми вышеописанными свойствами лексического окружения.

## В ES6 существует одно различие между компонентами LexicalEnvironment и VariableEnvironment. Оно заключается в том, что первое используется для хранения объявлений функций и переменных, объявленных с помощью ключевых слов let и const, а второе — только для хранения привязок переменных, объявленных с использованием ключевого слова var.

## Рассмотрим примеры, иллюстрирующие то, что мы только что обсудили:

## 

## Схематичное представление контекста выполнения для этого кода будет выглядеть так:

## 

## Как вы, вероятно, заметили, переменные и константы, объявленные с помощью ключевых слов let и const, не имеют связанных с ними значений, а переменным, объявленным с помощью ключевого слова var, назначено значение undefined.

## Это так из-за того, что во время создания контекста в коде осуществляется поиск объявлений переменных и функций, при этом объявления функций целиком хранятся в окружении. Значения переменных, при использовании var, устанавливаются в undefined, а при использовании let или const остаются неинициализированными.

## Именно поэтому можно получить доступ к переменным, объявленным с помощью var, до их объявления (хотя они и будут иметь значение undefined), но, при попытке доступа к переменным или константам, объявленным с помощью let и const, выполняемой до их объявления, возникает ошибка.

## Только что мы только что описали, называется «поднятием переменных» (Hoisting). Объявления переменных «поднимаются» в верхнюю часть их лексической области видимости до выполнения операций присвоения им каких-либо значений.

## Стадия выполнения кода

## Это, пожалуй, самая простая часть данного материала. На этой стадии выполняется присвоение значений переменным и осуществляется выполнение кода.

## Обратите внимание на то, что если в процессе выполнения кода JS-движок не сможет найти в месте объявления значение переменной, объявленной с помощью ключевого слова let, он присвоит этой переменной значение undefined.