**Мы будем говорить о переменных let/const здесь**

* В этой статье мы будем использовать переменные let в примерах.
* Переменные, объявленные с помощью const, ведут себя так же, так что эта статья и о них.

Какая область видимости у переменных? Блочная

С помощью блоков {...} мы можем изолировать часть кода, выполняющую свою собственную задачу, с переменными, принадлежащими только ей:

{

// показать сообщение

let message = "Hello";

alert(message);

}

{

// показать другое сообщение

let message = "Goodbye";

alert(message);

}

То же самое можно сказать и про циклы for и while:

for (let i = 0; i < 3; i++) {

// переменная i видна только внутри for

alert(i); // 0, потом 1, потом 2

}

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

Визуально let i = 0; находится вне блока кода {...}, однако здесь в случае с for есть особенность: переменная, объявленная внутри (...), считается частью блока.

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.

Ниже, makeCounter создает функцию «счётчик», которая при каждом вызове возвращает следующее число:

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

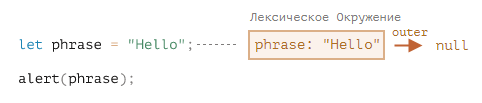
В JavaScript у каждой выполняемой функции, блока кода {...} и скрипта есть связанный с ними внутренний (скрытый) объект, называемый лексическим окружением LexicalEnvironment.

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».**

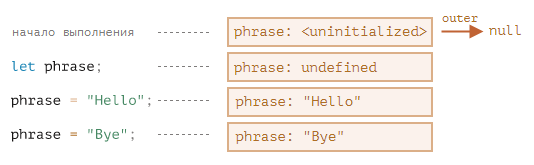
Например, в этом простом коде только одно лексическое окружение:



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.

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



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

1. При запуске скрипта лексическое окружение предварительно заполняется всеми объявленными переменными.
   * Изначально они находятся в состоянии «Uninitialized». Это особое внутреннее состояние, которое означает, что движок знает о переменной, но на нее нельзя ссылаться, пока она не будет объявлена с помощью let. Это почти то же самое, как если бы переменная не существовала.
2. Появляется определение переменной let phrase. У неё ещё нет присвоенного значения, поэтому присваивается undefined. С этого момента мы можем использовать переменную.
3. Переменной phrase присваивается значение.
4. Переменная phrase меняет значение.

Пока что всё выглядит просто, правда?

* Переменная – это свойство специального внутреннего объекта, связанного с текущим выполняющимся блоком/функцией/скриптом.
* Работа с переменными – это на самом деле работа со свойствами этого объекта.

**Лексическое окружение – объект спецификации**

«Лексическое окружение» – это объект спецификации: он существует только «теоретически» в спецификации языка для описания того, как все работает. Мы не можем получить этот объект в нашем коде и манипулировать им напрямую.

JavaScript-движки также могут оптимизировать его, отбрасывать неиспользуемые переменные для экономии памяти и выполнять другие внутренние действия, но при этом видимое поведение остается таким, как описано.

**[Function Declaration](https://learn.javascript.ru/closure" \l "shag-2-function-declaration)**

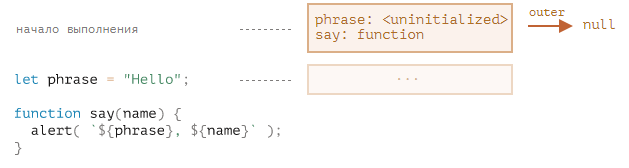
Функция – это тоже значение, как и переменная.

**Разница заключается в том, что Function Declaration мгновенно инициализируется полностью.**

Когда создается лексическое окружение, Function Declaration сразу же становится функцией, готовой к использованию (в отличие от let, который до момента объявления не может быть использован).

Именно поэтому мы можем вызвать функцию, объявленную как Function Declaration, до самого её объявления.

Вот, к примеру, начальное состояние глобального лексического окружения при добавлении функции:

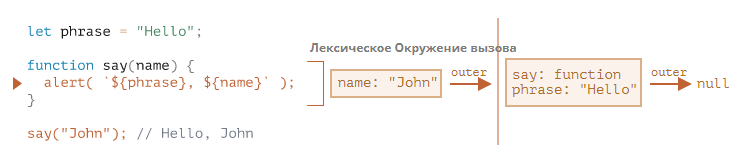


Конечно, такое поведение касается только Function Declaration, а не Function Expression, в которых мы присваиваем функцию переменной, например, let say = function(name) {...}.

### [Шаг 3. Внутреннее и внешнее лексическое окружение](https://learn.javascript.ru/closure" \l "shag-3-vnutrennee-i-vneshnee-leksicheskoe-okruzhenie)

Когда запускается функция, в начале ее вызова автоматически создается новое лексическое окружение для хранения локальных переменных и параметров вызова.

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



В процессе вызова функции у нас есть два лексических окружения: внутреннее (для вызываемой функции) и внешнее (глобальное):

* 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".

* Внешнее лексическое окружение – это глобальное лексическое окружение.

В нём находятся переменная phrase и сама функция.

У внутреннего лексического окружения есть ссылка на внешнее outer.

**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 строгом режиме (use strict). Без строгого режима, для обратной совместимости, присваивание несуществующей переменной создаёт новую глобальную переменную с таким же именем.

Все функции помнят лексическое окружение, в котором они были созданы. Технически здесь нет никакой магии: все функции имеют скрытое свойство [[Environment]], которое хранит ссылку на лексическое окружение, в котором была создана функция

Так функция запоминает, где она была создана, независимо от того, где она вызывается. Ссылка на [[Environment]] устанавливается один раз и навсегда при создании функции.

[Замыкание](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BC%D1%8B%D0%BA%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) – это функция, которая запоминает свои внешние переменные и может получить к ним доступ.

Но, как было описано выше, в JavaScript, все функции изначально являются замыканиями (есть только одно исключение, про которое будет рассказано в [Синтаксис "new Function"](https://learn.javascript.ru/new-function)).

То есть они автоматически запоминают, где были созданы, с помощью скрытого свойства [[Environment]], и все они могут получить доступ к внешним переменным.

Почитать вики.

## [Сборка мусора](https://learn.javascript.ru/closure" \l "sborka-musora)

Обычно лексическое окружение удаляется из памяти вместе со всеми переменными после завершения вызова функции. Это связано с тем, что на него нет ссылок. Как и любой объект JavaScript, оно хранится в памяти только до тех пор, пока к нему можно обратиться.

Однако если существует вложенная функция, которая все еще доступна после завершения функции, то она имеет свойство [[Environment]], ссылающееся на лексическое окружение.

В этом случае лексическое окружение остается доступным даже после завершения работы функции.

Например:

function f() {

let value = 123;

return function() {

alert(value);

}

}

let g = f(); // g.[[Environment]] хранит ссылку на лексическое окружение

// из соответствующего вызова f()

Обратите внимание, что если f() вызывается много раз и результирующие функции сохраняются, то все соответствующие объекты лексического окружения также будут сохранены в памяти. В приведенном ниже коде – все три:

function f() {

let value = Math.random();

return function() { alert(value); };

}

// 3 функции в массиве, каждая из которых ссылается на лексическое окружение

// из соответствующего вызова f()

let arr = [f(), f(), f()];

Объект лексического окружения исчезает, когда становится недоступным (как и любой другой объект). Другими словами, он существует только до тех пор, пока на него ссылается хотя бы одна вложенная функция.

В приведенном ниже коде после удаления вложенной функции ее окружающее лексическое окружение (а значит, и value) очищается из памяти:

function f() {

let value = 123;

return function() {

alert(value);

}

}

let g = f(); // пока существует функция g, value остается в памяти

g = null; // ...и теперь память очищена.

Функция sayHi использует имя внешней переменной. Какое значение будет использоваться при выполнении функции?

let name = "John";

function sayHi() {

alert("Hi, " + name);

}

name = "Pete";

sayHi(); // что будет показано: "John" или "Pete"?

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?

Ответ: **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.

### [Независимы ли счётчики?](https://learn.javascript.ru/closure" \l "nezavisimy-li-schyotchiki)

важность: 5

Здесь мы делаем два счётчика: counter и counter2, используя одну и ту же функцию makeCounter.

Они независимы? Что покажет второй счётчик? 0,1 или 2,3 или что-то ещё?

function makeCounter() {

let count = 0;

return function() {

return count++;

};

}

let counter = makeCounter();

let counter2 = makeCounter();

alert( counter() ); // 0

alert( counter() ); // 1

alert( counter2() ); // ?

alert( counter2() ); // ?

решение

Ответ: **0,1.**

Функции counter и counter2 созданы разными вызовами makeCounter.

Так что у них независимые внешние лексические окружения, у каждого из которых свой собственный count.

### [Видна ли переменная?](https://learn.javascript.ru/closure" \l "vidna-li-peremennaya)

важность: 4

Что выведет данный код?

let x = 1;

function func() {

console.log(x); // ?

let x = 2;

}

func();

P.S. В этой задаче есть подвох. Решение не очевидно.

решение

Ответ: **ошибка**.

Попробуйте запустить этот код:

let x = 1;

function func() {

console.log(x); // ReferenceError: Cannot access 'x' before initialization

let x = 2;

}

func();

В этом примере мы можем наблюдать характерную разницу между «несуществующей» и «неинициализированной» («uninitialized») переменной.

Как вы могли прочитать в статье [Область видимости переменных, замыкание](https://learn.javascript.ru/closure), переменная находится в «неинициализированном» («uninitialized») состоянии с момента входа в блок кода (или функцию). И остается неинициализированной до соответствующего оператора let.

Другими словами, переменная технически существует, но не может быть использована до let.

Приведенный выше код демонстрирует это.

function func() {

// локальная переменная x известна движку с самого начала выполнения функции,

// но она неинициализированна ("uninitialized") до let ("мёртвая зона")

// следовательно, ошибка

console.log(x); // ReferenceError: Cannot access 'x' before initialization

let x = 2;

}

Эту зону временной непригодности переменной (от начала блока кода до let) иногда называют «мёртвой зоной».