

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Пара вопросов

Лексическое Окружение

Вложенные функции

Окружение в деталях


Блоки кода и циклы, IIFE

Сборка мусора

Задачи (7)

Комментарии

Поделиться

[Редактировать на GitHub](#)[🏠 → Язык программирования JavaScript](#)
[→ Продвинутая работа с функциями](#) 8-го августа 2020

Замыкание

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

Мы знаем, что функция может получить доступ к переменным из внешнего окружения, эта возможность используется очень часто.

Но что произойдёт, когда внешние переменные изменятся? Функция получит последнее значение или то, которое существовало на момент создания функции?

И что произойдёт, когда функция переместится в другое место в коде и будет вызвана оттуда – получит ли она доступ к внешним переменным своего нового местоположения?

Разные языки ведут себя по-разному в таких случаях, и в этой главе мы рассмотрим поведение JavaScript.

Пара вопросов

Для начала давайте рассмотрим две ситуации, а затем изучим внутренние механизмы шаг за шагом, чтобы вы смогли ответить на эти и более сложные вопросы в будущем.

1. Функция `sayHi` использует внешнюю переменную `name`. Какое значение будет использовать функция при выполнении?

```
1 let name = "John";
2
3 function sayHi() {
4   alert("Hi, " + name);
5 }
6
7 name = "Pete";
8
9 sayHi(); // что будет показано: "John" или "Pete"?
```

Такие ситуации распространены и в браузерной и в серверной разработке. Выполнение функции может быть запланировано позже, чем она была создана, например, после какого-нибудь пользовательского действия или сетевого запроса.

Итак, вопрос в том, получит ли она доступ к последним изменениям?

2. Функция `makeWorker` создаёт другую функцию и возвращает её. Новая функция может быть вызвана откуда-то ещё. Получит ли она доступ к внешним переменным из места своего создания или места выполнения или из обоих?

```
1 function makeWorker() {
2   let name = "Pete";
3
4   return function() {
5     alert(name);
6   };
7 }
8
9 let name = "John";
10
11 // create a function
12 let work = makeWorker();
```

```
13
14 // call it
15 work(); // что будет показано? "Pete" (из места созда
```

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Пара вопросов

Лексическое Окружение

Вложенные функции

Окружение в деталях

Блоки кода и циклы, IIFE

Сборка мусора

Задачи (7)

Комментарии

Поделиться



Редактировать на GitHub



Лексическое Окружение

Чтобы понять, что происходит, давайте для начала обсудим, что такое «переменная» на самом деле.

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:

```
let phrase = "Hello"; ----- LexicalEnvironment
                                phrase: "Hello" outer → null
alert(phrase);
```

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`.

А вот как оно изменяется при объявлении и присваивании переменной:

```
начало выполнения ----- <пусто> outer → null
let phrase; ----- phrase: undefined
phrase = "Hello"; ----- phrase: "Hello"
phrase = "Bye"; ----- phrase: "Bye"
```

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.

Пока что всё выглядит просто, правда?

Итого:

- 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.

Function Declaration

До сих пор мы рассматривали только переменные. Теперь рассмотрим `Function Declaration`.

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Пара вопросов

Лексическое Окружение

Вложенные функции

Окружение в деталях

Блоки кода и циклы, IIFE

Сборка мусора

Задачи (7)

Комментарии

Поделиться



[Редактировать на GitHub](#)



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 мы можем 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`:

начало выполнения

```
let phrase = "Hello";  
function say(name) {  
  alert( `${phrase}, ${name}` );  
}
```

say: function
outer → null

say: function
phrase: "Hello"

Внутреннее и внешнее лексическое окружение

Теперь давайте продолжим и посмотрим, что происходит, когда функция получает доступ к внешней переменной.

В течение вызова `say()` использует внешнюю переменную `phrase`. Давайте разберёмся подробно, что происходит.

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):

<

```
let phrase = "Hello";  
function say(name) {  
  alert( `${phrase}, ${name}` );  
}  
say("John"); // Hello, John
```

LexicalEnvironment вызова

name: "John" outer →

say: function
phrase: "Hello" outer → null

>

Итак, 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 `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.

Давайте посмотрим, как происходит поиск в нашем примере:

- Когда `alert` внутри `say` хочет получить доступ к `name`, он немедленно находит переменную в лексическом окружении функции.
- Когда он хочет получить доступ к `phrase`, которой нет локально, он следует дальше по ссылке к внешнему лексическому окружению и находит переменную там.

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Пара вопросов

Лексическое окружение

Вложенные функции

Окружение в деталях

Блоки кода и циклы, IIFE

Сборка мусора

Задачи (7)

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
let phrase = "Hello";

function say(name) {
  alert( `${phrase}, ${name}` );
}

say("John"); // Hello, John
```

Теперь у нас есть ответ на первый вопрос из начала главы.

Функция получает текущее значение внешних переменных, то есть, их последнее значение

Старые значения переменных нигде не сохраняются. Когда функция хочет получить доступ к переменной, она берёт её текущее значение из своего или внешнего лексического окружения.

Так что, ответ на первый вопрос: Pete :

```
1 let name = "John";
2
3 function sayHi() {
4   alert("Hi, " + name);
5 }
6
7 name = "Pete"; // (*)
8
9 sayHi(); // Pete
```

Порядок выполнения кода, приведённого выше:

1. В глобальном лексическом окружении есть `name: "John"`.
2. На строке `(*)` глобальная переменная изменяется, теперь `name: "Pete"`.
3. Момент, когда выполняется функция `sayHi()` и берёт переменную `name` извне. Теперь из глобального лексического окружения, где переменная уже равна `"Pete"`.

i Один вызов – одно лексическое окружение

Пожалуйста, обратите внимание, что новое лексическое окружение функции создаётся каждый раз, когда функция выполняется.

И, если функция вызывается несколько раз, то для каждого вызова будет своё лексическое окружение, со своими, специфичными для этого вызова, локальными переменными и параметрами.

i Лексическое окружение – это специальный внутренний объект

«Лексическое окружение» – это специальный внутренний объект. Мы не можем получить его в нашем коде и изменять напрямую. Сам движок JavaScript может оптимизировать его, уничтожать неиспользуемые переменные для освобождения памяти и выполнять другие внутренние уловки, но видимое поведение объекта должно оставаться таким, как было описано.

Вложенные функции

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

Это очень легко сделать в JavaScript.

Мы можем использовать это для упорядочивания нашего кода, например, как здесь:

```
1 function sayHiBye(firstName, lastName) {
2
3   // функция-помощник, которую мы используем ниже
```

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Пара вопросов

Лексическое Окружение

Вложенные функции

Окружение в деталях

Блоки кода и циклы, IIFE

Сборка мусора

Задачи (7)

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
4 function getFullName() {
5     return firstName + " " + lastName;
6 }
7
8 alert( "Hello, " + getFullName() );
9 alert( "Bye, " + getFullName() );
10
11 }
```

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.

Что ещё интереснее, 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.

Например, здесь, вложенная функция присваивается новому объекту в [конструкторе](#):

```
1 // функция-конструктор возвращает новый объект
2 function User(name) {
3
4     // методом объекта становится вложенная функция
5     this.sayHi = function() {
6         alert(name);
7     };
8 }
9
10 let user = new User("John");
11 user.sayHi(); // у кода метода "sayHi" есть доступ к вн
```



А здесь мы просто создаём и возвращаем функцию «счётчик»:



```
1 function makeCounter() {
2     let count = 0;
3
4     return function() {
5         return count++; // есть доступ к внешней переменной
6     };
7 }
8
9 let counter = makeCounter();
10
11 alert( counter() ); // 0
12 alert( counter() ); // 1
13 alert( counter() ); // 2
```

Давайте продолжим с примером `makeCounter`. Он создаёт функцию «counter», которая возвращает следующее число при каждом вызове. Несмотря на простоту, немного модифицированные варианты этого кода применяются на практике, например, в [генераторе псевдослучайных чисел](#) и во многих других случаях.

Как же это работает изнутри?

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

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}
```

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Пара вопросов

Лексическое Окружение

Вложенные функции

Окружение в деталях

Блоки кода и циклы, IIFE

Сборка мусора

Задачи (7)

Комментарии

Поделиться



[Редактировать на GitHub](#)



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

В этом примере `count` будет найден на шаге 2. Когда внешняя переменная модифицируется, она изменится там, где была найдена. Значит, `count++` найдёт внешнюю переменную и увеличит её значение в лексическом окружении, которому она принадлежит. Как если бы у нас было `let count = 1`.

Теперь рассмотрим два вопроса:

1. Можем ли мы каким-нибудь образом сбросить счётчик `count` из кода, который не принадлежит `makeCounter`? Например, после вызова `alert` в коде выше.
2. Если мы вызываем `makeCounter` несколько раз – нам возвращается много функций `counter`. Они независимы или разделяют одну и ту же переменную `count`?

Попробуйте ответить на эти вопросы перед тем, как продолжить чтение.

...

Готовы?

Хорошо, давайте ответим на вопросы.

1. Такой возможности нет: `count` – локальная переменная функции, мы не можем получить к ней доступ извне.
2. Для каждого вызова `makeCounter()` создаётся новое лексическое окружение функции, со своим собственным `count`. Так что, получившиеся функции `counter` – независимы.

Вот демо:

```
1 function makeCounter() {
2   let count = 0;
3   return function() {
4     return count++;
5   };
6 }
7
8 let counter1 = makeCounter();
9 let counter2 = makeCounter();
10
11 alert( counter1() ); // 0
12 alert( counter1() ); // 1
13
14 alert( counter2() ); // 0 (независимо)
```

Надеюсь, ситуация с внешними переменными теперь ясна. Для большинства ситуаций такого понимания вполне достаточно, но в спецификации есть ряд деталей, которые мы, для простоты, опустили. Далее мы разберём происходящее ещё более подробно.

Окружение в деталях

Вот что происходит в примере с `makeCounter` шаг за шагом. Пройдите их, чтобы убедиться, что вы разобрались с каждой деталью.

Пожалуйста, обратите внимание на дополнительное свойство `[[Environment]]`, про которое здесь рассказано. Мы не упоминали о нём раньше для простоты.

1. Когда скрипт только начинает выполняться, есть только глобальное лексическое окружение:

```
▶ function makeCounter() { [[Environment]] →
  let count = 0;

  return function() {
    return count++;
  };
}
...
makeCounter: function → outer null
```



В этот начальный момент есть только функция `makeCounter`, потому что это Function Declaration. Она ещё не выполняется.

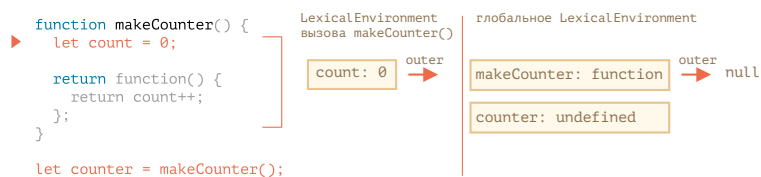
Все функции «при рождении» получают скрытое свойство `[[Environment]]`, которое ссылается на лексическое окружение места, где они были созданы.

Мы ещё не говорили об этом, это то, каким образом функции знают, где они были созданы.

В данном случае, `makeCounter` создан в глобальном лексическом окружении, так что `[[Environment]]` содержит ссылку на него.

Другими словами, функция навсегда запоминает ссылку на лексическое окружение, где она была создана. И `[[Environment]]` – скрытое свойство функции, которое содержит эту ссылку.

- Код продолжает выполняться, объявляется новая глобальная переменная `counter`, которой присваивается результат вызова `makeCounter`. Вот снимок момента, когда интерпретатор находится на первой строке внутри `makeCounter()`:



В момент вызова `makeCounter()` создаётся лексическое окружение, для хранения его переменных и аргументов.

Как и все лексические окружения, оно содержит две вещи:

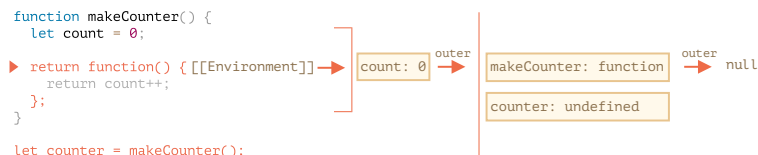
- Environment Record с локальными переменными. В нашем случае `count` – единственная локальная переменная (появляющаяся, когда выполняется строчка с `let count`).
- Ссылка на внешнее окружение, которая устанавливается в значение `[[Environment]]` функции. В данном случае, `[[Environment]]` функции `makeCounter` ссылается на глобальное лексическое окружение.

Итак, теперь у нас есть два лексических окружения: первое – глобальное, второе – для текущего вызова `makeCounter`, с внешней ссылкой на глобальный объект.

- В процессе выполнения `makeCounter()` создаётся небольшая вложенная функция.

Не имеет значения, какой способ объявления функции используется: Function Declaration или Function Expression. Все функции получают свойство `[[Environment]]`, которое ссылается на лексическое окружение, в котором они были созданы. То же самое происходит и с нашей новой маленькой функцией.

Для нашей новой вложенной функции значением `[[Environment]]` будет текущее лексическое окружение `makeCounter()` (где она была создана):



Пожалуйста, обратите внимание, что на этом шаге внутренняя функция была создана, но ещё не вызвана. Код внутри `function() { return count++; }` не выполняется.

- Выполнение продолжается, вызов `makeCounter()` завершается, и результат (небольшая вложенная функция) присваивается глобальной переменной `counter`:



Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Пара вопросов

Лексическое Окружение

Вложенные функции

Окружение в деталях

Блоки кода и циклы, IIFE

Сборка мусора

Задачи (7)

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
    return count++;  
  };  
}  
  
let counter = makeCounter();  
  
alert( counter() );
```

counter: function

В этой функции есть только одна строчка: `return count++`, которая будет выполнена, когда мы вызовем функцию.

5. При вызове `counter()` для этого вызова создаётся новое лексическое окружение. Оно пустое, так как в самом `counter` локальных переменных нет. Но `[[Environment]]` `counter` используется, как ссылка на внешнее лексическое окружение `outer`, которое даёт доступ к переменным предшествующего вызова `makeCounter`, где `counter` был создан.

```
function makeCounter() {  
  let count = 0;  
  
  return function() {  
    return count++;  
  };  
}  
  
let counter = makeCounter();  
  
alert( counter() );
```

[[Environment]]

outer: {
 count: 0
}

outer: {
 makeCounter: function
 counter: function
}

outer: null

Теперь, когда вызов ищет переменную `count`, он сначала ищет в собственном лексическом окружении (пустое), а затем в лексическом окружении предшествующего вызова `makeCounter()`, где и находит её.

Пожалуйста, обратите внимание, как здесь работает управление памятью. Хотя `makeCounter()` закончил выполнение некоторое время назад, его лексическое окружение остаётся в памяти, потому что есть вложенная функция с `[[Environment]]`, который ссылается на него.

В большинстве случаев, объект лексического окружения существует до того момента, пока есть функция, которая может его использовать. И только тогда, когда таких не остаётся, окружение уничтожается.

6. Вызов `counter()` не только возвращает значение `count`, но также увеличивает его. Обратите внимание, что модификация происходит «на месте». Значение `count` изменяется конкретно в том окружении, где оно было найдено.

```
function makeCounter() {  
  let count = 0;  
  
  return function() {  
    return count++;  
  };  
}  
  
let counter = makeCounter();  
  
alert( counter() );  
alert( counter() );
```

изменено здесь

outer: {
 count: 1
}

outer: {
 makeCounter: function
 counter: function
}

outer: null

7. Следующие вызовы `counter()` сделают то же самое.

Теперь ответ на второй вопрос из начала главы должен быть очевиден.

Функция `work()` в коде ниже получает `name` из того места, где была создана, через ссылку на внешнее лексическое окружение:

```
function makeWorker() {  
  let name = "Pete";  
  
  return function() {  
    alert(name);  
  };  
}  
  
let name = "John";  
let work = makeWorker();  
  
work();
```

outer: {
 name: "Pete"
}

outer: {
 makeWorker: function
 name: "John"
}

outer: null

Так что, результатом будет `"Pete"`.

Но, если бы в `makeWorker()` не было `let name`, тогда бы поиск продолжился дальше и была бы взята глобальная переменная, как мы видим из приведённой выше цепочки. В таком случае, результатом было бы `"John"`.

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Пара вопросов

Лексическое Окружение

Вложенные функции

Окружение в деталях

Блоки кода и циклы, IIFE

Сборка мусора

Задачи (7)

Комментарии

Поделиться



[Редактировать на GitHub](#)



Замыкания

В программировании есть общий термин: «замыкание», – которое должен знать каждый разработчик.

Замыкание – это функция, которая запоминает свои внешние переменные и может получить к ним доступ. В некоторых языках это невозможно, или функция должна быть написана специальным образом, чтобы получилось замыкание. Но, как было описано выше, в JavaScript, все функции изначально являются замыканиями (есть только одно исключение, про которое будет рассказано в [Синтаксис "new Function"](#)).

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

Когда на собеседовании фронтенд-разработчик получает вопрос: «что такое замыкание?», – правильным ответом будет определение замыкания и объяснения того факта, что все функции в JavaScript являются замыканиями, и, может быть, несколько слов о технических деталях: свойстве `[[Environment]]` и о том, как работает лексическое окружение.

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. Вот пара примеров.

If

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

```
let phrase = "Hello";

if (true) {
  let user = "John";
  alert(`${phrase}, ${user}`);
}

alert(user); // Error, no such variable!
```

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 и Function Expression, 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:

```
1 for (let i = 0; i < 10; i++) {
2   // У каждой итерации цикла своё собственное лексическое
3   // {i: value}
4 }
5
6 alert(i); // Ошибка, нет такой переменной
```

Обратите внимание: `let i` визуально находится снаружи `{...}`. Но конструкция `for` – особенная в этом смысле, у каждой итерации цикла своё собственное лексическое окружение с текущим `i` в нём.

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Пара вопросов

Лексическое Окружение

Вложенные функции

Окружение в деталях

Блоки кода и циклы, IIFE

Сборка мусора

Задачи (7)

Комментарии

Поделиться



[Редактировать на GitHub](#)



И так же, как и в `if`, ниже цикла `i` невидима.

Блоки кода

Мы также можем 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 мы создадим 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.

Это может произойти, если название переменной – широко распространённое слово, а авторы скрипта не знают друг о друге.

Если мы хотим этого избежать, мы можем использовать блок кода для изоляции всего скрипта или какой-то его части:

```
1 {  
2   // сделать какую-нибудь работу с локальными переменными  
3  
4   let message = "Hello";  
5  
6   alert(message); // Hello  
7 }  
8  
9 alert(message); // Ошибка: переменная message не опреде.
```

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 мы должны использовать сегодня, но, так как вы можете встретить это в старых скриптах, полезно понимать принцип работы.

IIFE выглядит так:

```
1 (function() {  
2  
3   let message = "Hello";  
4  
5   alert(message); // Hello  
6  
7 })();
```

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

Function Expression обернуто 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:

```
1 // Попробуйте объявить и сразу же вызвать функцию  
2 function() { // <-- Error: Unexpected token (  
3  
4   let message = "Hello";  
5  
6   alert(message); // Hello  
7
```

```
}());
```

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Пара вопросов

Лексическое Окружение

Вложенные функции

Окружение в деталях

Блоки кода и циклы, IIFE

Сборка мусора

Задачи (7)

Комментарии

Поделиться


[Редактировать на GitHub](#)


Даже если мы скажем: «хорошо, давайте добавим имя», – это не сработает, потому что JavaScript не позволяет вызывать Function Declaration немедленно.

```
1 // ошибка синтаксиса из-за скобок ниже
2 function go() {
3
4 }(); // <-- не можете вызывать Function Declaration нем
```

Так что, скобки вокруг функции – это трюк, который позволяет показать JavaScript, что функция была создана в контексте другого выражения, и, таким образом, это функциональное выражение: ей не нужно имя и её можно вызвать немедленно.

Кроме скобок, существуют и другие пути показать JavaScript, что мы имеем в виду Function Expression:

```
1 // Пути создания IIFE
2
3 (function() {
4   alert("Скобки вокруг функции");
5 })();
6
7 (function() {
8   alert("Скобки вокруг всего");
9 })();
10
11 !function() {
12   alert("Выражение начинается с логического оператора N
13 });
14
15 +function() {
16   alert("Выражение начинается с унарного плюса");
17 }();
```

Во всех перечисленных случаях мы объявляем Function Expression и немедленно выполняем его. Ещё раз заметим, что в настоящий момент нет необходимости писать подобный код.

Сборка мусора

Обычно лексическое окружение очищается и удаляется после того, как функция выполнялась. Например:

```
1 function f() {
2   let value1 = 123;
3   let value2 = 456;
4 }
5
6 f();
```

Здесь два значения, которые технически являются свойствами лексического окружения. Но после того, как `f()` завершится, это лексическое окружение станет недоступно, поэтому оно удалится из памяти.

...Но, если есть вложенная функция, которая всё ещё доступна после выполнения `f`, то у неё есть свойство `[[Environment]]`, которое ссылается на внешнее лексическое окружение, тем самым оставляя его достижимым, «живым»:

```
1 function f() {
2   let value = 123;
```

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Пара вопросов

Лексическое Окружение

Вложенные функции

Окружение в деталях

Блоки кода и циклы, IIFE

Сборка мусора

Задачи (7)

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
3
4  function g() { alert(value); }
5
6  return g;
7  }
8
9  let g = f(); // g доступно и продолжает держать внешнее
```

Обратите внимание, если `f()` вызывается несколько раз и возвращаемые функции сохраняются, тогда все соответствующие объекты лексического окружения продолжают держаться в памяти. Вот три такие функции в коде ниже:

```
1  function f() {
2    let value = Math.random();
3
4    return function() { alert(value); };
5  }
6
7  // три функции в массиве, каждая из них ссылается на ле
8  // из соответствующего вызова f()
9  let arr = [f(), f(), f()];
```

Объект лексического окружения умирает, когда становится недоступным (как и любой другой объект). Другими словами, он существует только до того момента, пока есть хотя бы одна вложенная функция, которая ссылается на него.

В следующем коде, после того как `g` станет недоступным, лексическое окружение функции (и, соответственно, `value`) будет удалено из памяти;

```
1  function f() {
2    let value = 123;
3
4    function g() { alert(value); }
5
6    return g;
7  }
8
9  let g = f(); // пока g существует,
10 // соответствующее лексическое окружение существует
11
12 g = null; // ...а теперь память очищается
```

Оптимизация на практике

Как мы видели, в теории, пока функция жива, все внешние переменные тоже сохраняются.

Но на практике движки JavaScript пытаются это оптимизировать. Они анализируют использование переменных и, если легко по коду понять, что внешняя переменная не используется – она удаляется.

Одним из важных побочных эффектов в V8 (Chrome, Opera) является то, что такая переменная становится недоступной при отладке.

Попробуйте запустить следующий пример в Chrome с открытой Developer Tools.

Когда код будет поставлен на паузу, напишите в консоли `alert(value)`.

```
1  function f() {
2    let value = Math.random();
3
4    function g() {
5      debugger; // в консоли: напишите alert(value); Тако
6    }
7  }
```

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Пара вопросов

Лексическое Окружение

Вложенные функции

Окружение в деталях

Блоки кода и циклы, IIFE

Сборка мусора

Задачи (7)

Комментарии

Поделиться



[Редактировать на GitHub](#)



```
8   return g;
9 }
10
11 let g = f();
12 g();
```

Как вы можете видеть – такой переменной не существует! В теории, она должна быть доступна, но попала под оптимизацию движка.

Это может приводить к забавным (если удаётся решить быстро) проблемам при отладке. Одна из них – мы можем увидеть не ту внешнюю переменную при совпадающих названиях:

```
1 let value = "Сюрприз!";
2
3 function f() {
4   let value = "ближайшее значение";
5
6   function g() {
7     debugger; // в консоли: напишите alert(value); Сюрприз!
8   }
9
10  return g;
11 }
12
13 let g = f();
14 g();
```

⚠ До встречи!

Эту особенность V8 полезно знать. Если вы занимаетесь отладкой в Chrome/Opera, рано или поздно вы с ней встретитесь.

Это не баг в отладчике, а скорее особенность V8. Возможно со временем это изменится. Вы всегда можете проверить это, запустив пример на этой странице.

✓ Задачи

Независимы ли счётчики? [🔗](#)

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?

```
1 function makeCounter() {
2   let count = 0;
3
4   return function() {
5     return count++;
6   };
7 }
8
9 let counter = makeCounter();
10 let counter2 = makeCounter();
11
12 alert( counter() ); // 0
13 alert( counter() ); // 1
14
15 alert( counter2() ); // ?
16 alert( counter2() ); // ?
```

решение

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Пара вопросов

Лексическое окружение

Вложенные функции

Окружение в деталях

Блоки кода и циклы, IIFE

Сборка мусора

Задачи (7)

Комментарии

Поделиться



[Редактировать на GitHub](#)



Объект счётчика [↗](#)

важность: 5

Здесь объект счётчика создан с помощью функции-конструктора.

Будет ли он работать? Что покажет?

```
1 function Counter() {
2   let count = 0;
3
4   this.up = function() {
5     return ++count;
6   };
7   this.down = function() {
8     return --count;
9   };
10 }
11
12 let counter = new Counter();
13
14 alert( counter.up() ); // ?
15 alert( counter.up() ); // ?
16 alert( counter.down() ); // ?
```

решение

Функция в if [↗](#)

Посмотрите на код. Какой будет результат у вызова на последней строке?

```
1 let phrase = "Hello";
2
3 if (true) {
4   let user = "John";
5
6   function sayHi() {
7     alert(`${phrase}, ${user}`);
8   }
9 }
10
11 sayHi();
```

решение

Сумма с помощью замыканий [↗](#)

важность: 4

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

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

Например:

```
1 sum(1)(2) = 3
2 sum(5)(-1) = 4
```

решение

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Пара вопросов

Лексическое Окружение

Вложенные функции

Окружение в деталях

Блоки кода и циклы, IIFE

Сборка мусора

Задачи (7)

Комментарии

Поделиться



[Редактировать на GitHub](#)



Фильтрация с помощью функции

важность: 5

У нас 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:

```
1 /* .. ваш код для inBetween и inArray */
2 let arr = [1, 2, 3, 4, 5, 6, 7];
3
4 alert( arr.filter(inBetween(3, 6)) ); // 3,4,5,6
5
6 alert( arr.filter(inArray([1, 2, 10])) ); // 1,2
```

[Открыть песочницу с тестами для задачи.](#)

решение



Сортировать по полю

важность: 5

У нас есть массив объектов, который нужно отсортировать:

```
1 let users = [
2   { name: "John", age: 20, surname: "Johnson" },
3   { name: "Pete", age: 18, surname: "Peterson" },
4   { name: "Ann", age: 19, surname: "Hathaway" }
5  ];
```

Обычный способ был бы таким:

```
1 // по имени (Ann, John, Pete)
2 users.sort((a, b) => a.name > b.name ? 1 : -1);
3
4 // по возрасту (Pete, Ann, John)
5 users.sort((a, b) => a.age > b.age ? 1 : -1);
```

Можем ли мы сделать его короче, скажем, вот таким?

```
1 users.sort(byField('name'));
2 users.sort(byField('age'));
```

То есть, чтобы вместо функции, мы просто писали `byField(fieldName)`.

Напишите функцию `byField`, которая может быть использована для этого.

решение



важность: 5

Следующий код создаёт массив из стрелков (shooters).

Каждая функция предназначена выводить их порядковые номера. Но что-то пошло не так...



```
1 function makeArmy() {
2   let shooters = [];
3
4   let i = 0;
5   while (i < 10) {
6     let shooter = function() { // функция shooter
7       alert( i ); // должна выводить порядковый номер
8     };
9     shooters.push(shooter);
10    i++;
11  }
12
13  return shooters;
14 }
15
16 let army = makeArmy();
17
18 army[0](); // у 0-го стрелка будет номер 10
19 army[5](); // и у 5-го стрелка тоже будет номер 10
20 // ... у всех стрелков будет номер 10, вместо 0, 1, 2, ...
```

Почему у всех стрелков одинаковые номера? Почините код, чтобы он работал как задумано.

[Открыть песочницу с тестами для задачи.](#)

решение



Проводим [курсы по JavaScript и фреймворкам](#). ✕

Комментарии

перед тем как писать...

Раздел

[Продвинутая работа с функциями](#)

Навигация по уроку

Пара вопросов

Лексическое окружение

Вложенные функции

Окружение в деталях

Блоки кода и циклы, IIFE

Сборка мусора

Задачи (7)

Комментарии

Поделиться



[Редактировать на GitHub](#)