9 августа 2023 г.

Область видимости переменных, 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.

Мы будем говорить о переменных `let/const` здесь

В JavaScript существует три способа объявить переменную: `let`, `const` (современные), и `var` (пережиток прошлого).

- 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.
- Старые переменные `var` имеют несколько характерных отличий, они будут рассмотрены в главе Устаревшее ключевое слово "var".

Блоки кода

Если переменная объявлена внутри блока кода `{...}`, то она видна только внутри этого блока.

Например:

```
{
  // выполняем некоторые действия с локальной переменной, которые не должны бы

  let message = "Hello"; // переменная видна только в этом блоке

  alert(message); // Hello
}

alert(message); // ReferenceError: message is not defined
```

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);
}
```

Без блоков была бы ошибка

Обратите внимание, что без отдельных блоков возникнет ошибка, если мы используем let с существующим именем переменной:

```
// показать сообщение
let message = "Hello";
alert(message);

// показать другое сообщение
let message = "Goodbye"; // SyntaxError: Identifier 'message' has already
alert(message);
```

Для `if`, `for`, `while` и т.д. переменные, объявленные в блоке кода `{...}`, также видны только внутри:

```
if (true) {
  let phrase = "Hello";

  alert(phrase); // Hello
}

alert(phrase); // Ошибка, нет такой переменной!
```

В этом случае после завершения работы `if` нижний `alert` не увидит `phrase`, что и приведет к ошибке.

И это замечательно, поскольку это позволяет нам создавать блочно-локальные переменные, относящиеся только к ветви `if`.

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.

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:

```
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.

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.

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
```

Несмотря на простоту этого примера, немного модифицированные его варианты применяются на практике, например, в генераторе псевдослучайных чисел и во многих других случаях.

Как это работает? Если мы создадим несколько таких счётчиков, будут ли они независимыми друг от друга? Что происходит с переменными?

Понимание таких вещей полезно для повышения общего уровня владения JavaScript и для более сложных сценариев. Так что давайте немного углубимся.

Лексическое окружение

> **Здесь водятся драконы!**
>
> Глубокое техническое описание – впереди.
>
> Как бы мне ни хотелось избежать низкоуровневых деталей языка, любое представление о JavaScript без них будет недостаточным и неполным, так что приготовьтесь.

Для большей наглядности объяснение разбито на несколько шагов.

Шаг 1. Переменные

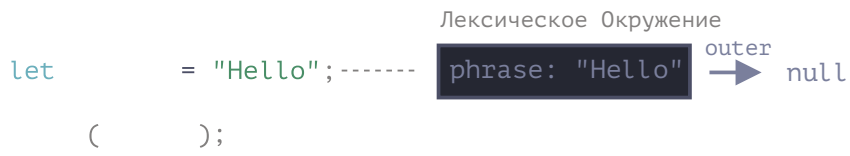
В JavaScript у каждой выполняемой функции, блока кода `{...}` и скрипта есть связанный с ними внутренний (скрытый) объект, называемый *лексическим окружением* `LexicalEnvironment`.

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:

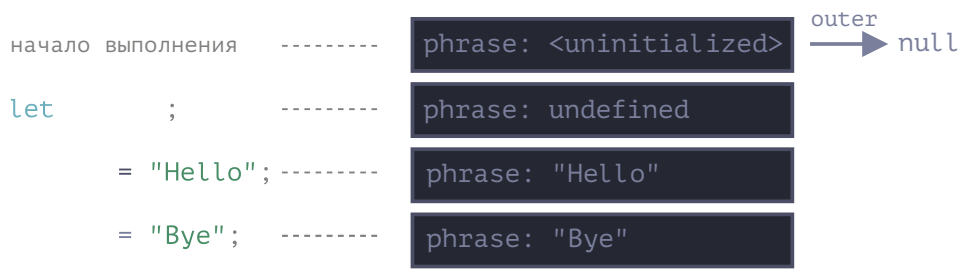


Это, так называемое, глобальное лексическое окружение, связанное со всем скриптом.

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.

Вот более длинный код:



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.

Шаг 2. Function Declaration

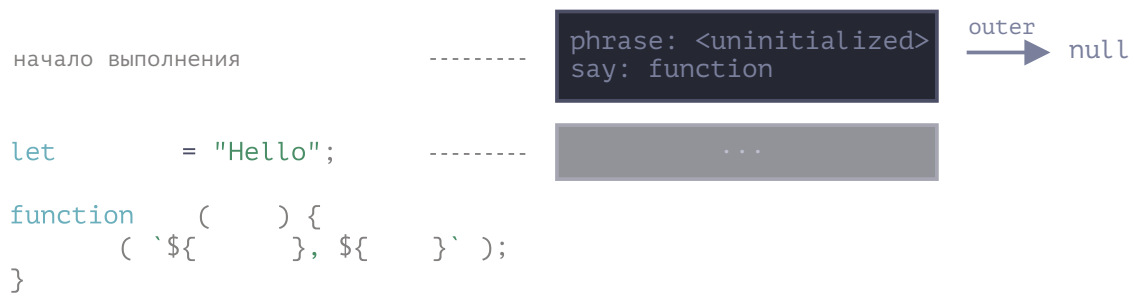
Функция – это тоже значение, как и переменная.

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:

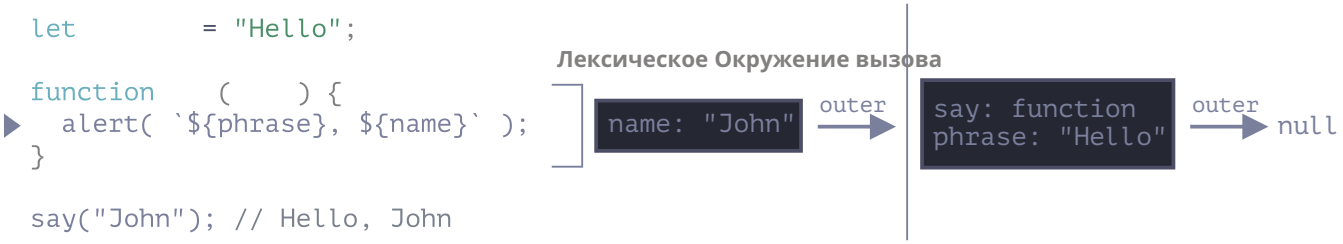


Конечно, такое поведение касается только Function Declaration, а не Function Expression, в которых мы присваиваем функцию переменной, например, `let say = function(name) {...}`.

Шаг 3. Внутреннее и внешнее лексическое окружение

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.

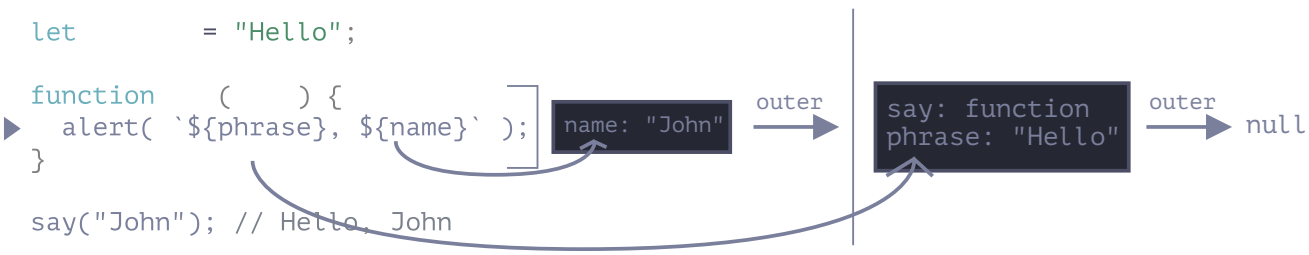
У внутреннего лексического окружения есть ссылка на внешнее `outer`.

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.

Давайте посмотрим, как происходит поиск в нашем примере:

- Для переменной `name`, `alert` внутри `say` сразу же находит ее во внутреннем лексическом окружении.
- Когда `alert` хочет получить доступ к `phrase`, он не находит её локально, поэтому вынужден обратиться к внешнему лексическому окружению и находит `phrase` там.



Шаг 4. Возврат функции

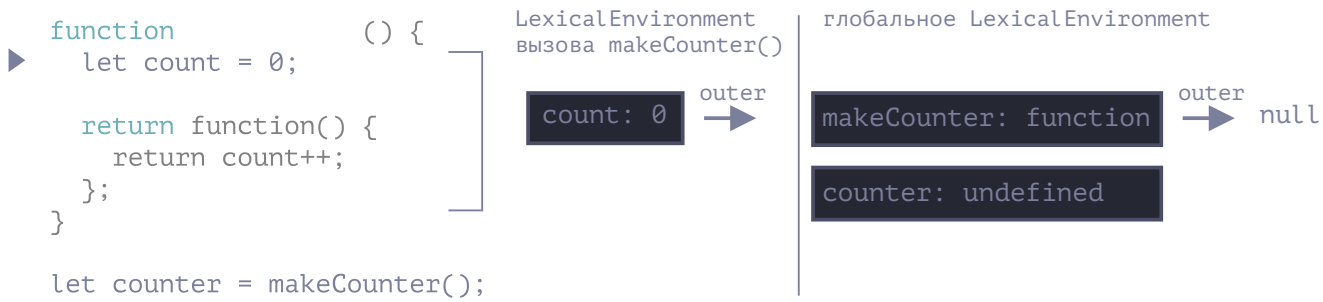
Давайте вернёмся к примеру с `makeCounter`:

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

```
3
4     return function() {
5       return count++;
6     };
7   }
8
9   let counter = makeCounter();
```

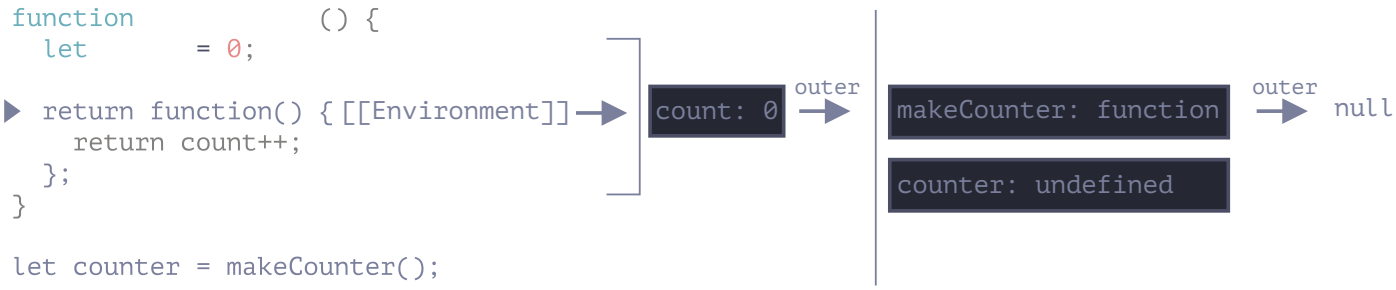
В начале каждого вызова `makeCounter()` создается новый объект лексического окружения, в котором хранятся переменные для конкретного запуска `makeCounter`.

Таким образом, мы имеем два вложенных лексических окружения, как в примере выше:



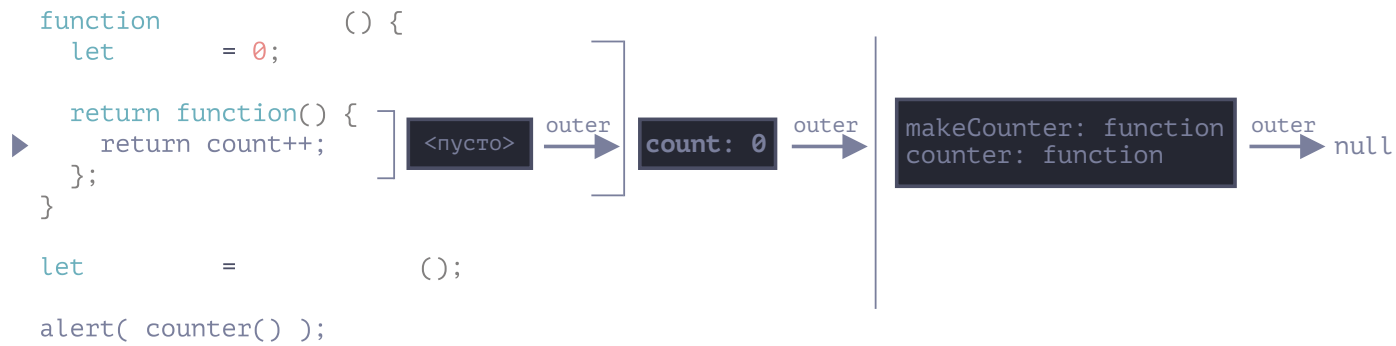
Отличие заключается в том, что во время выполнения `makeCounter()` создается крошечная вложенная функция, состоящая всего из одной строки: `return count++`. Мы ее еще не запускаем, а только создаем.

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:



Таким образом, `counter.[[Environment]]` имеет ссылку на `{count: 0}` лексического окружения. Так функция запоминает, где она была создана, независимо от того, где она вызывается. Ссылка на `[[Environment]]` устанавливается один раз и навсегда при создании функции.

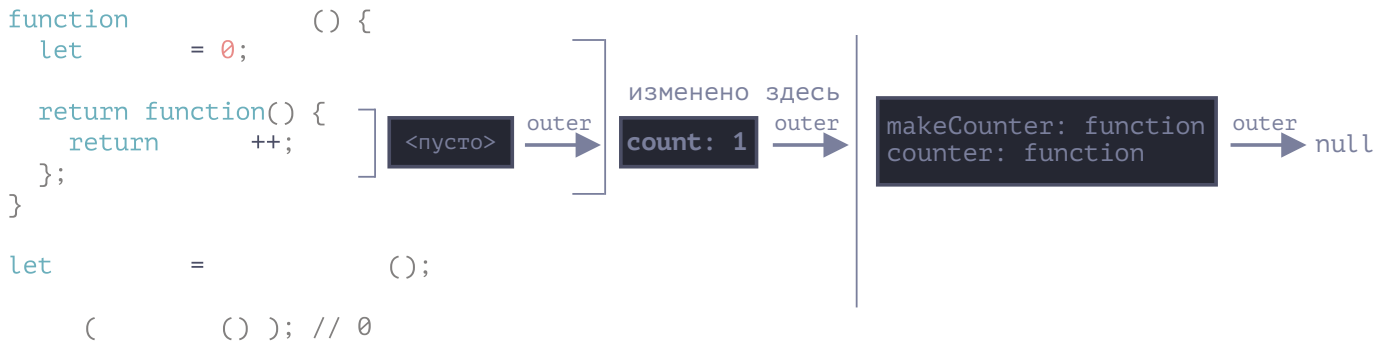
Впоследствии, при вызове `counter()`, для этого вызова создается новое лексическое окружение, а его внешняя ссылка на лексическое окружение берется из `counter.[[Environment]]`:



Теперь, когда код внутри `counter()` ищет переменную `count`, он сначала ищет ее в собственном лексическом окружении (пустом, так как там нет локальных переменных), а затем в лексическом окружении внешнего вызова `makeCounter()`, где находит `count` и изменяет ее.

Переменная обновляется в том лексическом окружении, в котором она существует.

Вот состояние после выполнения:



Если мы вызовем `counter()` несколько раз, то в одном и том же месте переменная `count` будет увеличена до `2`, `3` и т.д.

> **Замыкания**
>
> В программировании есть общий термин: «замыкание», – который 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-разработчику задают 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.

Сборка мусора

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:

```
1  function f() {
2    let value = 123;
3
4    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.
```

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, Edge, 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)`.

```
function f() {
  let value = Math.random();

  function g() {
    debugger; // в консоли: напишите alert(value); Такой переменной нет!
  }

  return g;
}

let g = f();
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:

```
let value = "Сюрприз!";

function f() {
  let value = "ближайшее значение";

  function g() {
    debugger; // в консоли: напишите alert(value); Сюрприз!
  }

  return g;
}

let g = f();
g();
```

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

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

Задачи

Учитывает ли функция последние изменения?

важность: 5

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"?
```

Такие ситуации встречаются как при разработке для браузера, так и для сервера. 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?

решение

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.

Какие переменные доступны?

важность: 5

Приведенная ниже функция `makeWorker` создает другую функцию и возвращает ее. Эта новая функция может быть вызвана из другого места.

Будет ли она иметь доступ к внешним переменным из места своего создания, или из места вызова, или из обоих мест?

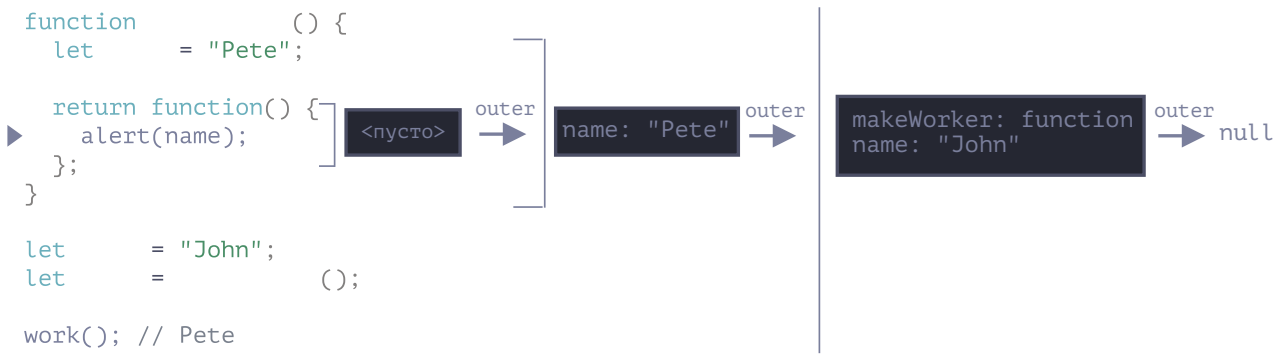
```
function makeWorker() {
  let name = "Pete";

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

let name = "John";

// создаём функцию
let work = makeWorker();

// вызываем её
work(); // что будет показано?
```

Какое значение будет показано? «Pete» или «John»?

решение

Ответ: **Pete**.

Функция `work()` в приведенном ниже коде получает `name` из места его происхождения через ссылку на внешнее лексическое окружение:



Таким образом, в результате мы получаем `"Pete"`.

Но если бы в `makeWorker()` не было `let name`, то поиск шел бы снаружи и брал глобальную переменную, что мы видим из приведенной выше цепочки. В этом случае результатом было бы `"John"`.

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?

```
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`.

Объект счётчика

важность: 5

Здесь объект счётчика создан с помощью функции-конструктора.

Будет ли он работать? Что покажет?

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

  this.up = function() {
    return ++count;
  };
  this.down = function() {
    return --count;
  };
}

let counter = new Counter();

alert( counter.up() ); // ?
alert( counter.up() ); // ?
alert( counter.down() ); // ?
```

решение

Несомненно, он отлично будет работать.

Обе вложенные функции были созданы с одним и тем же внешним лексическим окружением, так что они имеют доступ к одной и той же переменной `count`:

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

  this.up = function() {
    return ++count;
  };

  this.down = function() {
    return --count;
  };
```

```
}

let counter = new Counter();

alert( counter.up() ); // 1
alert( counter.up() ); // 2
alert( counter.down() ); // 1
```

Функция внутри if

важность: 5

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

```
let phrase = "Hello";

if (true) {
  let user = "John";

  function sayHi() {
    alert(`${phrase}, ${user}`);
  }
}

sayHi();
```

решение

Результатом будет **ошибка**.

Функция `sayHi` объявлена внутри `if`, так что она живёт только внутри этого блока. Снаружи нет `sayHi`.

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:

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

решение

Чтобы вторые скобки заработали, первые – должны вернуть функцию.

Вот так:

```
function sum(a) {

  return function(b) {
    return a + b; // берёт "a" из внешнего лексического окружения
  };

}

alert( sum(1)(2) ); // 3
alert( sum(5)(-1) ); // 4
```

Видна ли переменная?

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 initiali
  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, 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
  // но она неинициализированна ("uninitialized") до let ("мёртвая зон
  // следовательно, ошибка

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

  let x = 2;
}
```

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

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:

```
/* .. ваш код для inBetween и inArray */
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
```

Открыть песочницу с тестами для задачи.

решение

Фильтр inBetween

```
function inBetween(a, b) {
  return function(x) {
    return x >= a && x <= b;
  };
}

let arr = [1, 2, 3, 4, 5, 6, 7];
alert( arr.filter(inBetween(3, 6)) ); // 3,4,5,6
```

Фильтр inArray

```
function inArray(arr) {
  return function(x) {
    return arr.includes(x);
  };
}

let arr = [1, 2, 3, 4, 5, 6, 7];
alert( arr.filter(inArray([1, 2, 10])) ); // 1,2
```

Открыть решение с тестами в песочнице.

Сортировать по полю

важность: 5

У нас 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);
```

Можем ли мы сделать его короче, например вот таким?

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

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

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

Открыть песочницу с тестами для задачи.

решение

```
function byField(fieldName){
  return (a, b) => a[fieldName] > b[fieldName] ? 1 : -1;
}
```

Открыть решение с тестами в песочнице.

Армия функций

важность: 5

Следующий код создаёт массив из стрелков (`shooters`).

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

```
function makeArmy() {
  let shooters = [];

  let i = 0;
  while (i < 10) {
    let shooter = function() { // функция shooter
      alert( i ); // должна выводить порядковый номер
    };
    shooters.push(shooter); // и добавлять стрелка в массив
    i++;
  }

  // ...а в конце вернуть массив из всех стрелков
  return shooters;
}

let army = makeArmy();

// все стрелки выводят 10 вместо их порядковых номеров (0, 1, 2, 3...)
army[0](); // 10 от стрелка с порядковым номером 0
army[1](); // 10 от стрелка с порядковым номером 1
army[2](); // 10 ...и т.д.
```

Почему у всех стрелков одинаковые номера?

Почините код, чтобы он работал как задумано.

Открыть песочницу с тестами для задачи.

решение

Давайте посмотрим, что происходит внутри `makeArmy`, и решение станет очевидным.

1.

Она создаёт пустой массив `shooters`:

```
let shooters = [];
```

2.

В цикле заполняет его `shooters.push(function...)`.

Каждый элемент – это функция, так что получится такой массив:

```
shooters = [
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); },
  function () { alert(i); }
];
```

3.

Функция возвращает массив.

Позже вызов `army[5]()` получит элемент `army[5]` из массива (это будет функция) и вызовет её.

Теперь, почему все эти функции показывают одно и то же?

Всё потому, что внутри функций `shooter` нет локальной переменной `i`. Когда вызывается такая функция, она берёт `i` из своего внешнего лексического окружения.

Какое будет значение у `i`?

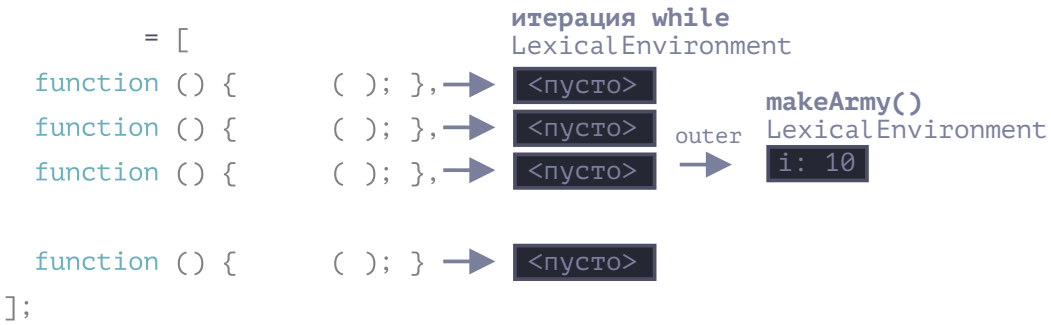
Если мы посмотрим в исходный код:

```
function makeArmy() {
  ...
  let i = 0;
  while (i < 10) {
```

```
    let shooter = function() { // функция shooter
      alert( i ); // должна выводить порядковый номер
    };
    shooters.push(shooter); // и добавлять стрелка в массив
    i++;
  }
  ...
}
```

...Мы увидим, что оно живёт в лексическом окружении, связанном с текущим вызовом `makeArmy()`. Но, когда вызывается `army[5]()`, `makeArmy` уже завершила свою работу, и последнее значение `i`: `10` (конец цикла `while`).

Как результат, все функции `shooter` получат одно и то же значение из внешнего лексического окружения: последнее значение `i=10`.



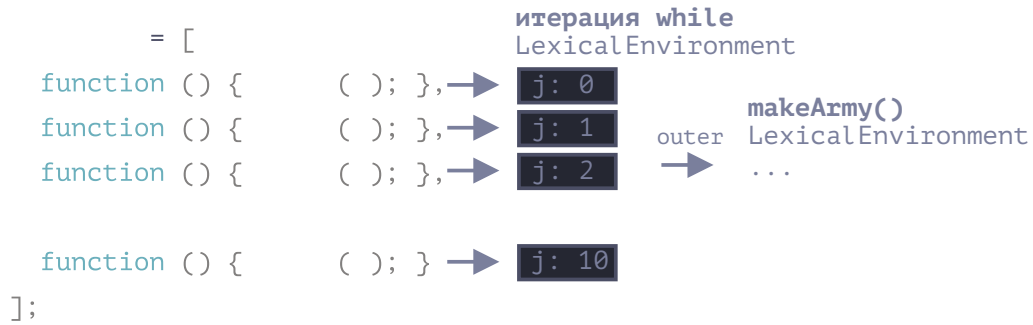
Как вы можете видеть выше, на каждой итерации блока `while {...}` создается новое лексическое окружение. Чтобы исправить это, мы можем скопировать значение `i` в переменную внутри блока `while {...}`, например, так:

```
function makeArmy() {
  let shooters = [];

  let i = 0;
  while (i < 10) {
      let j = i;
      let shooter = function() { // функция shooter
        alert( j ); // должна выводить порядковый номер
      };
    shooters.push(shooter);
    i++;
  }

  return shooters;
}

let army = makeArmy();

// теперь код работает правильно
army[0](); // 0
army[5](); // 5
```

Здесь `let j = i` объявляет «итерационно-локальную» переменную `j` и копирует в нее `i`. Примитивы копируются «по значению», поэтому фактически мы получаем независимую копию `i`,

принадлежащую текущей итерации цикла.

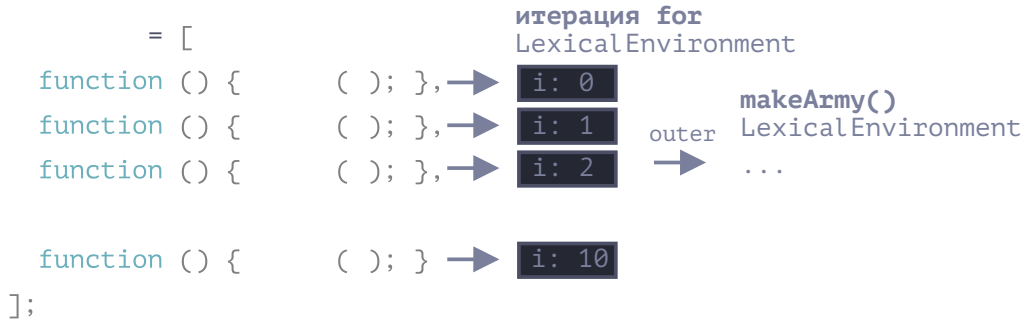
Функции `shooter` работают правильно, потому что значение `i` теперь живет чуть ближе. Не в лексическом окружении `makeArmy()`, а в лексическом окружении, соответствующем текущей итерации цикла:



Этой проблемы также можно было бы избежать, если бы мы использовали `for` в начале, например, так:

```
1  function makeArmy() {
2
3    let shooters = [];
4
5    for (let i = 0; i < 10; i++) {
6      let shooter = function() { // функция shooter
7        alert( i ); // должна выводить порядковый номер
8      };
9      shooters.push(shooter);
10   }
11
12   return shooters;
13 }
14
15 let army = makeArmy();
16
17 army[0](); // 0
18 army[5](); // 5
```

По сути, это то же самое, поскольку `for` на каждой итерации создает новое лексическое окружение со своей переменной `i`. Поэтому функция `shooter`, создаваемая на каждой итерации, ссылается на свою собственную переменную `i`, причем именно с этой итерации.



Теперь, когда вы приложили столько усилий, чтобы прочитать это объяснение, а конечный вариант оказался так прост – использовать `for`, вы можете задаться вопросом – стоило ли оно того?

Что ж, если бы вы могли легко ответить на вопрос из задачи, вы бы не стали читать решение. Так что, должно быть, эта задача помогла вам лучше понять суть дела.

Кроме того, действительно встречаются случаи, когда человек предпочитает `while`, а не `for`, и другие сценарии, где такие проблемы реальны.

Открыть решение с тестами в песочнице.

Предыдущий урок | Следующий урок

Поделиться

Карта учебника

Комментарии

- Если вам кажется, что в статье что-то не так - вместо комментария напишите на GitHub.
- Для одной строки кода используйте тег `<code>`, для нескольких строк кода — тег `<pre>`, если больше 10 строк — ссылку на песочницу (plnkr, JSBin, codepen...)
- Если что-то непонятно в статье — пишите, что именно и с какого места.

Miron
месяц назад

Почему такой код не работает как надо?

```
function makeArmy() {
let shooters = [];

let i = 0;
while (i < 10) {
let shooter = function() {
let shooter_number = i;
console.log( shooter_number );
};
shooters.push(shooter); // и добавлять стрелка в массив
i++;
}

// ...а в конце вернуть массив из всех стрелков
return shooters;
}
```


Михаил Саввин → Miron
18 дней назад

Всё дело в том, что shooter_number создаётся при вызове функции, а не при её создании. В самой функции нет переменной i, поэтому каждая функция - shooter смотрит итерацию while, не находит while и там, и поднимается до окружения makeArmy - i уже равно 10

Sosockol → Miron
месяц назад

Потому что здесь нужно передать i в аргумент функции, что бы это стало локальной переменной. Тогда при обращении к этой функции она не будет выходить в родительское окружение.

Вот так должно работать:

```
while (i < 10) {
let shooter = function(i) {
let shooter_number = i;
console.log( shooter_number );
};
```

0 0 Ответить

В **Влад**
2 месяца назад

Понравилась последняя задача , не думал что разница for и while так глубока

2 0 Ответить

Д **Данила**
2 месяца назад

В задаче "Функция внутри if", я вроде как неправильно ответил, решил что у функции не будет проблем с поиском переменных и нам выведется строка Hello, John
Но в ответе было написано, что нам выведет ошибку. Я удивился, вбил программу в первый попавшийся компилятор JS, и мне вывело в консоли строку Hello, John. Как такое возможно?

0 0 Ответить

Maxim → Данила
месяц назад

Тут 2 поведения.

С ошибкой это если в начале программы будет директива "use strict" (в решении пропустили этот нюанс), если директивы нет то получается так что функция доступна вне лексического блока.

Я кстати говоря объяснял этот момент здесь, вдруг поможет.

2 0 Ответить

Aleksandras → Maxim
месяц назад

> С ошибкой это если в начале программы будет директива "use strict" (в решении *пропустили этот нюанс*)

Все примеры кода в учебнике подразумевают использование строгого режима, за редкими исключениями, когда присутствует определённое пояснение к коду, говорящее об ином. Об этом было сказано в самом начале первой главы, во втором разделе — статья «Строгий режим "use strict"».

0 0 Ответить

Maxim → Aleksandras
месяц назад

А что если человек просто открыл главу и решает задачки? Не все же проходят и досконально помнят все нюансы учебника.

Ну и потом я уже детально объяснял разницу между двумя режимами.

2 0 Ответить

Александр Низовцев
2 месяца назад edited

Статья про LE, честно говоря, не очень. Если уж взялись объяснять подробно про [[Environment]] и прочее, сделали бы наглядное пошаговое руководство, когда, почему и что появляется, шаг за шагом. А то создается впечатление, что вы это и попытались сделать, но часть шагов просто пропустили, подразумевая, что человек это уже должен сам понять. Новичка эта статья с первого прочтения только запутает, а не поможет разобраться.

0 0 Ответить

Джун на фронте: (Тг @divatoz)
3 месяца назад

Выучили JavaScript и что же дальше? Как стать веб-разработчиком? Фриланс или офис? Нужны ли фреймворки?

💡Ответы на эти и другие вопросы вы найдёте в моём **телеграм-блоге «Джун на фронте»**, где я **где я уже 1000 часов** осваиваю веб-разработку с нуля (я не понимал return у функций) и до оффера!

😫 Так как я тот ещё прокрастинатор, то **мой путь занимает 700 дней**, но как говорится, у самурая нет цели, есть только путь.

🍺 За это время я успел поработать верстальщиком, веб-дизайнером, фрилансером, изучил Vue с Nuxt, но вакансий на этом стеке очень мало, и **сейчас я изучаю React** по бесплатным источникам.

Я **НЕ ПРОДАЮ ОБУЧЕНИЕ, НЕ МЕНТОРЮ и НЕ РЕКЛАМИРУЮ КУРСЫ**. Я хочу собрать комьюнити единомышленников и вместе понять, чего же хочет рынок от ИТ-специалистов в России!

0 1 Ответить

Klishe
3 месяца назад

Скажите пожалуйста, почему мы можем выводить переменную-счётчик цикла for в таком виде:
for (let i = 5; i < 10; i++) console.log(i);
Но при этом эта переменная недоступна где-либо ещё? Обратите внимаение, я не использовал фигурные скобки блока инструкций, а следовательно нет блочно-локальной области видимости. Или я не прав? И что такое вообще находится в круглых скобках у цикла for? Инструкции? Выражения? Параметры?

0 0 Ответить

Anonym → Klishe
3 месяца назад

Читайте статью урока внимательно!

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

То есть это означает, что переменнные объявленные внутри "круглых скобок" for -> () также считаются частью блока.

После того как цикл завершается, все переменные объявленные внутри круглых скобок for и фигурных скобок удаляются из памяти, и к ним больше нет доступа, именно поэтому к переменной i

нельзя обратиться за пределами самого цикла.

Чтобы можно было использовать переменую i после цикла, для этого достаточно объявить переменную выше цикла.

Например:

```
let i;

for (i = 5; i < 10; i++) {
        console.log(i);
```

показать больше

0 0 Ответить

K **Klishe** Anonym
3 месяца назад

А каком блоке идёт речь, если в моём примере я его даже не создал? У меня в примере: for (let i = 5; i < 10; i++) console.log(i); просто есть инструкция console.log(i) которая выполняется на каждой итерации цикла, а не блок инструкций {console.log(i)}.

Нашёл ответ от Maxim в другом разделе: /*
Конструкция for (не in/of) может создавать разное поведение:

1) Когда вы используете var для объявления переменных.
2) Когда вы используете let/const для переменных.

Собственно два эти случая порождают разные цепочки Environment Record.

Итак выполнение for делится на 2 этапа, это выполнение головы и выполнение тела.
За голову отвечает все то что находится в for(...), за тело все то что находится после круглых скобок.

Алгоритм выполнения головы зависит от того var или let/const вы там объявляете.
Если вы пишите var, то алгоритм головы никаких Environment Record не создает, он считает что объявление var для него такое же если бы вы этот var сделали до самой конструкции for. Но вот если у вас объявление let/const тогда алгоритм головы создает Environment Record, чтобы записать

показать больше

0 0 Ответить

A **Anonym** Klishe
3 месяца назад

-> "У меня в примере: for (let i = 5; i < 10; i++) console.log(i); просто есть инструкция console.log(i) которая выполняется на каждой итерации цикла, а не блок инструкций {console.log(i)}"
Даже если в теле цикла указана одна инструкция, она все равно относится к телу цикла, и это никак не меняет суть поведения.

Если переменная объявленна спомощью let внутри круглых скобок, то это переменная становится частью самого цикла, и за пределами она будет уже недоступна.
Поэтому имеет ли цикл вообще тело или нет, это все равно не влияет на область видимости переменной, хочешь чтобы переменная была доступна и после цикла, то объяви её до начало цикла и тогда её область видимости рассширится за пределы цикла.

Нужно внимательно изучать все уроки, там всё прекрасно доступным языком описано.

0 0 Ответить

А **Александр**
3 месяца назад

Подписка на Грецкий Фронтенд @gretskiy_frontend - это бесплатное код ревью до 500 строк. Подробнее спрашивайте у @as_krt 😉

0 1 Ответить

К **Константин**
4 месяца назад

задача "Функция внутри if" так как 'use strict' не указан , функция sayHi() выполнится без ошибок, так как функция объявленная с помощью function declaration всплывет в глобальную область видимости , и будет доступна в любом месте . Для того чтобы доступность sayHi() была ограничена любым блоком {} нужен строгий режим .

1 0 Ответить

Aleksandras → Константин
месяц назад edited

Все примеры кода в учебнике подразумевают использование строгого режима, за редкими исключениями, когда перед кодом уточняется, что будет описано иное поведение — об этом говорилось ещё в статье «Строгий режим — "use strict"», во втором разделе первой главы. Поэтому, в задаче ошибок нет.

0 0 Ответить

frontforme
5 месяцев назад

Учишь теорию и хочешь стать frontend-разработчиком?! Так держать!!

🎯 В своем блоге я рассказываю **как устроится в крупную компанию** frontend-разработчиком самым коротким путем. Давай познакомимся!

👤 Кто я такой?
Автор канала — ответственный за сервис **Frontend-специалист** из крупной компании.

Заходи:
tg: @davay_v_frontend

0 1 Ответить

М **Михаил Иванов**
5 месяцев назад

Если я могу решить задачу, я заглядываю в ответы, чтобы проверить, а так конечно спасибо за интересный пример)

1 0 Ответить

J **Jane Versus**
5 месяцев назад

хорошо, что переписали статью, теперь легче понять

0 Ответить

Загрузить ещё комментарии