Тема 3. Функции и замыкания

1. Глобальный объект.
2. Замыкания, внутренняя работа функции.
3. Конструкция new Function.
4. Объект функции.
5. Остаточные параметры и оператор расширения.
6. Каррирование.
7. Генераторы.
8. Колбэки.
9. Промис (Promise).
10. Промисификация.
11. Async/await.
12. Управление памятью в JavaScript.

Содержание данной темы включает материалы, доступные по адресу https://learn.javascript.ru.

1. **Глобальный объект**

Глобальный объект предоставляет переменные и функции, доступные в любом месте программы. По умолчанию это те, что встроены в язык или среду исполнения. В браузере он называется window, в Node.js – global, в другой среде исполнения может называться иначе. Недавно globalThis был добавлен в язык как стандартизированное имя для глобального объекта, которое должно поддерживаться в любом окружении. В некоторых браузерах (например, Edge) globalThis ещё не поддерживается, но легко реализуется с помощью полифила.

Прежде, решение было таким:

const getGlobal = function () {

if (typeof self !== undefined) {

return self;

}

if (typeof window !== undefined) {

return window;

}

if (typeof global !== undefined) {

return global;

}

throw new Error("unable to locate global object");

};

const globals = getGlobal();

// Сейчас есть `globalThis`

globalThis === window; // true

Рассмотрим подробнее глобальный объект window, так как наша среда – браузер. Ко всем свойствам глобального объекта можно обращаться напрямую:

alert("Привет");

// это то же самое, что и

window.alert("Привет");

В браузере глобальные функции и переменные, объявленные с помощью var (не let/const), становятся свойствами глобального объекта:

var gVar = 5;

alert(window.gVar); // 5 (становится свойством глобального объекта)

Такое поведение поддерживается для совместимости. В современных проектах, использующих [JavaScript-модули](https://learn.javascript.ru/modules), такого не происходит.

Если объявить переменную при помощи let, то такого не произойдет:

let gLet = 5;

alert(window.gLet); // undefined (не становится свойством глобального объекта)

Если свойство настолько важное, что надо сделать его доступным для всей программы, то запишите его в глобальный объект напрямую:

window.currentUser = {

name: "John"

};

alert(currentUser.name); // John

alert(window.currentUser.name); // John

При этом обычно не рекомендуется использовать глобальные переменные. Следует применять их как можно реже. Дизайн кода, при котором функция получает входные параметры и выдаёт определённый результат, чище, надёжнее и удобнее для тестирования, чем когда используются внешние, а тем более глобальные переменные.

Глобальный объект можно использовать, чтобы проверить поддержку современных возможностей языка. Например, проверить наличие встроенного объекта Promise (такая поддержка отсутствует в очень старых браузерах):

if (!window.Promise) {

alert("Ваш браузер очень старый!");

}

Если такой объект не поддерживается, то можно создать полифил: добавить функции, которые не поддерживаются окружением, но существуют в современном стандарте.

if (!window.Promise) {

window.Promise = ... // реализация современной возможности языка

}–

1. **Замыкания, внутренняя работа функции.**

JavaScript – язык с сильным функционально-ориентированным уклоном. Функция может быть динамически создана, скопирована в другую переменную или передана как аргумент другой функции и позже вызвана из совершенно другого места. Функция может получить доступ к переменным из внешнего окружения, эта возможность используется очень часто.

Рассмотрим две ситуации:

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

let name = "John";

function sayHi() {

alert("Hi, " + name);

}

name = "Pete";

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.

1. 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?

function makeWorker() {

let name = "Pete";

return function() {

alert(name);

};

}

let name = "John";

let work = makeWorker();

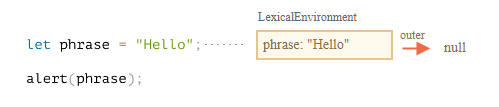
work(); // что будет показано? "Pete" (из места создания) или "John" (из места выполнения)

[**Лексическое окружение**](https://learn.javascript.ru/closure#leksicheskoe-okruzhenie)

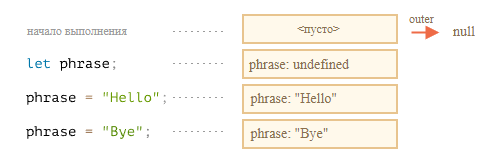
Чтобы понять, что происходит, давайте для начала, обсудим, что такое «переменная» на самом деле. В JavaScript у каждой выполняемой функции, блока кода и скрипта есть связанный с ними внутренний (скрытый) объект, называемый лексическим окружением LexicalEnvironment. Объект лексического окружения состоит 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:



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:



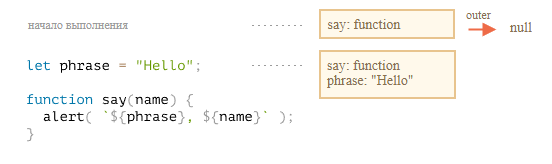
Прямоугольники с правой стороны демонстрируют, как глобальное лексическое окружение изменяется в процессе выполнения кода:

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.

[**Function Declaration**](https://learn.javascript.ru/closure#function-declaration)

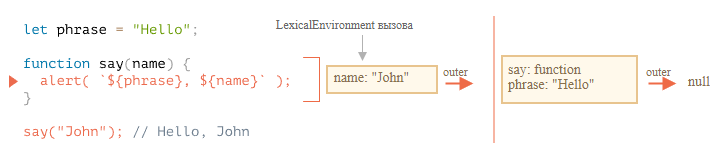
До сих поры рассматривались только переменные. Теперь рассмотрим Function Declaration. В отличие от переменных, объявленных с помощью let, они полностью инициализируются не тогда, когда выполнение доходит до них, а раньше, когда создаётся лексическое окружение. Для верхнеуровневых функций это означает момент, когда скрипт начинает выполнение. Вот почему можно 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:



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

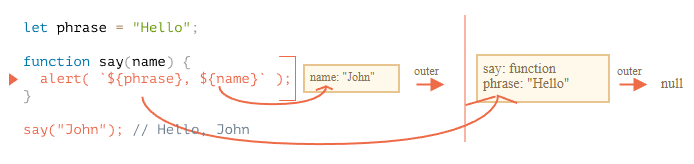
В течение вызова say() использует внешнюю переменную phrase. При запуске функции для неё автоматически создаётся новое лексическое окружение, для хранения локальных переменных и параметров вызова. Например, для say("John") это выглядит так (выполнение находится на строке, отмеченной стрелкой):



Итак, в процессе вызова функции 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. При вызове say("John") 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.

Рассмотрим, как происходит поиск в рассматриваемом 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.



Это ответ на первый вопрос из начала темы. 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:

let name = "John";

function sayHi() {

alert("Hi, " + name);

}

name = "Pete"; // (\*)

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. Выполняется функция sayHi() и берёт переменную name извне. Теперь из глобального лексического окружения, где переменная уже равна "Pete".

Обратите внимание, что новое лексическое окружение функции создаётся каждый раз, когда функция выполняется. Если функция вызывается несколько раз, то для каждого вызова будет своё лексическое окружение, со своими, специфичными для этого вызова, локальными переменными и параметрами.

Лексическое окружение – это специальный внутренний объект. Нельзя получить его в 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**](https://learn.javascript.ru/closure#vlozhennye-funktsii)

Функция называется вложенной, когда она создаётся внутри другой функции. Можно использовать это для упорядочивания кода, например:

function sayHiBye(firstName, lastName) {

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.

Вложенная функция может быть возвращена: либо в качестве свойства нового объекта, 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](https://learn.javascript.ru/constructor-new):

function User(name) {

this.sayHi = function() {

alert(name);

};

}

let user = new User("John");

user.sayHi();

Создаём и возвращаем функцию «счётчик»:

function makeCounter() {

let count = 0;

return function() {

return count++;

};

}

let counter = makeCounter();

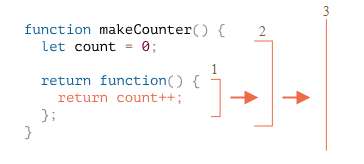
alert( counter() ); // 0

alert( counter() ); // 1

alert( counter() ); // 2

Рассмотрим далее 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](https://ru.wikipedia.org/wiki/%D0%93%D0%B5%D0%BD%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_%D0%BF%D1%81%D0%B5%D0%B2%D0%B4%D0%BE%D1%81%D0%BB%D1%83%D1%87%D0%B0%D0%B9%D0%BD%D1%8B%D1%85_%D1%87%D0%B8%D1%81%D0%B5%D0%BB) и во многих других случаях.

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



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 было let count = 1.

Теперь рассмотрим два вопроса:

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

Ответы на вопросы:

1. Такой возможности нет: count – локальная переменная функции, нельзя 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:

function makeCounter() {

let count = 0;

return function() {

return count++;

};

}

let counter1 = makeCounter();

let counter2 = makeCounter();

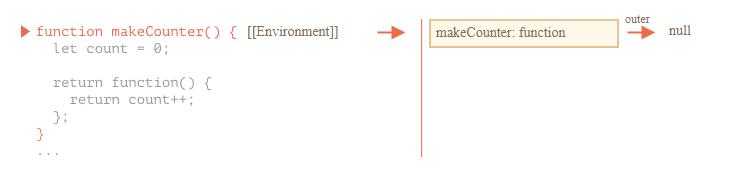
alert( counter1() ); // 0

alert( counter1() ); // 1

alert( counter2() ); // 0

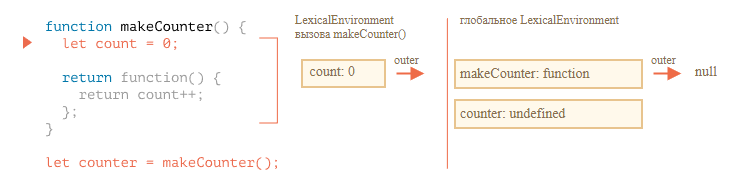
Рассмотрим, что происходит в примере с makeCounter шаг за шагом. Обратите внимание на дополнительное свойство [[Environment]].

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



В этот начальный момент есть только функция makeCounter, потому что это Function Declaration. Она ещё не выполняется. Все функции при создании получают скрытое свойство [[Environment]], которое ссылается на лексическое окружение места, 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.

1. Код продолжает выполняться, объявляется новая глобальная переменная counter, которой присваивается результат вызова makeCounter. На рисунке изображено состояние, когда интерпретатор находится на первой строке внутри makeCounter():



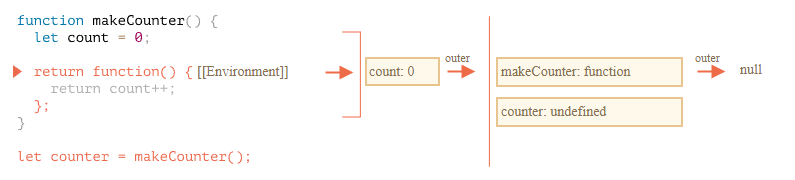
В момент вызова makeCounter() создаётся лексическое окружение, для хранения его переменных и аргументов. Как и все лексические окружения, оно содержит две вещи:

* Environment Record с локальными переменными. В рассматриваемом примере 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 есть два лексических окружения: первое – глобальное, второе – для текущего вызова makeCounter, с внешней ссылкой на глобальный объект.

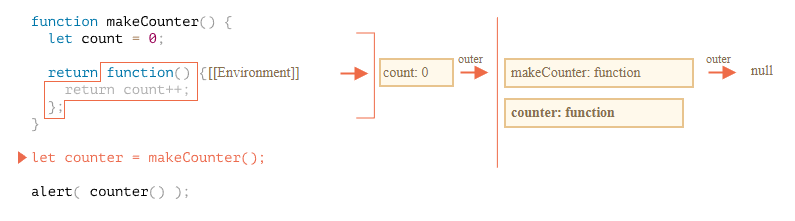
1. В процессе выполнения makeCounter() создаётся небольшая вложенная функция.

Не имеет значения, какой способ объявления функции используется: Function Declaration или Function Expression. Все функции получают свойство [[Environment]], которое ссылается на лексическое окружение, в которым они были созданы. Тоже происходит и с новой функцией. Для нее значением [[Environment]] будет текущее лексическое окружение makeCounter() (где она была создана):



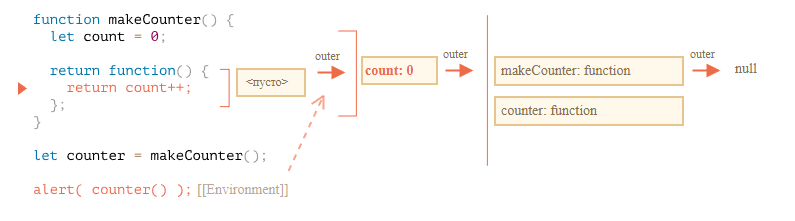
Обратите внимание, что на этом шаге внутренняя функция была создана, но ещё не вызвана. Код внутри function() { return count++ } не выполняется.

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



В этой функции есть только одна строчка: return count++, которая будет выполнена при вызове функции.

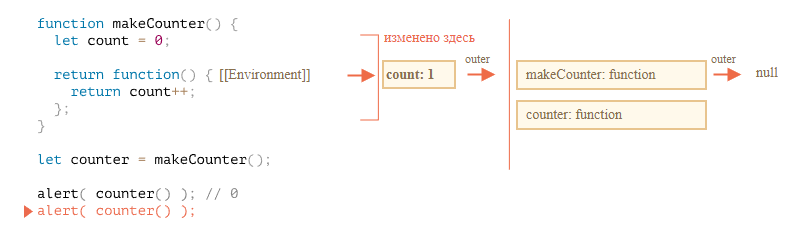
1. При вызове counter() для этого вызова создаётся новое лексическое окружение. Оно пустое, так как в самом counter локальных переменных нет. Но [[Environment]] counter используется, как ссылка на внешнее лексическое окружение outer, которое даёт доступ к переменным предшествующего вызова makeCounter,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



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.

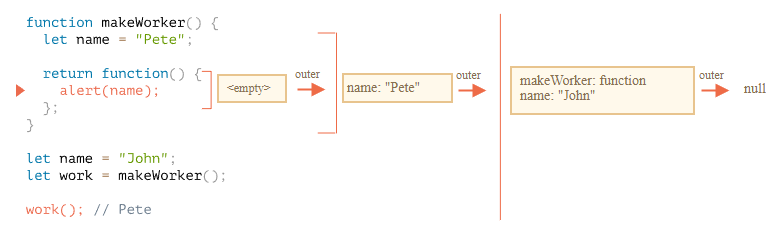
Обратите внимание, как здесь работает управление памятью. Хотя makeCounter() закончил выполнение некоторое время назад, его лексическое окружение остаётся в памяти, потому что есть вложенная функция с [[Environment]], который ссылается на него. В большинстве случаев, объект лексического окружения существует до того момента, пока есть функция, которая может его использовать. И только тогда, когда таких не остаётся, окружение уничтожается.

1. Вызов counter() не только возвращает значение count, но также увеличивает его. Модификация происходит «на месте». Значение count изменяется конкретно в том окружении, где оно было найдено.



1. Следующие вызовы counter() сделают то же самое.

Теперь ответ на второй вопрос из начала темы очевиден: функция work() в коде ниже получает name из того места, где была создана, через ссылку на внешнее лексическое окружение:



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. В таком случае, результатом было бы "John".

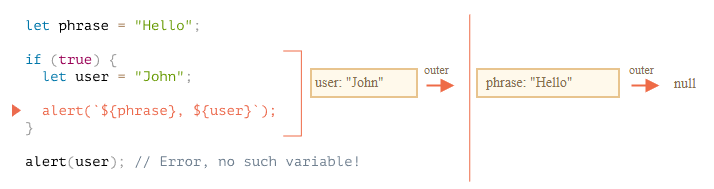
**Замыкания**

[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. 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"](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.

[**Блоки кода и циклы**](https://learn.javascript.ru/closure#bloki-koda-i-tsikly-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.

В следующем примере переменная user существует только в блоке if:



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.

Для цикла у каждой итерации своё отдельное лексическое окружение. Если переменная объявлена в for(let ...), то она также в нём:

for (let i = 0; i < 10; i++) {

// {i: value}

}

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

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

Также можно 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 создать глобальную переменную в одном скрипте, она станет доступна и в других. Но это становится источником конфликтов, если два скрипта используют одно и тоже имя переменной и перезаписывают друга друга. Это может произойти, если название переменной – широко распространённое слово, а авторы скрипта не знают друг о друге. Если надо этого избежать, то можно использовать блок кода для изоляции всего скрипта или какой-то его части:

{

let message = "Hello";

alert(message); // Hello

}

alert(message); // Ошибка: переменная message не определена

Из-за того, что у блока есть собственное лексическое окружение, код снаружи него (или в другом скрипте) не видит переменные этого блока.

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

function f() {

let value1 = 123;

let value2 = 456;

}

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:

function f() {

let value = 123;

function g() { alert(value); }

return g;

}

let g = f();

Но на практике движки JavaScript пытаются это оптимизировать. Они анализирует 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. Это может привести к тому, что можно 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);

}

return g;

}

let g = f();

g();

1. **Конструкция new Function.**

Существует ещё один вариант объявлять функции. Он используется крайне редко, но иногда другого решения не найти. Синтаксис для объявления функции:

let func = new Function([arg1, arg2, ...argN], functionBody);

Функция создается с заданными аргументами arg1...argN и телом functionBody. Это проще понять на конкретном примере. Здесь объявлена функция с двумя аргументами:

let sum = new Function('a', 'b', 'return a + b');

alert( sum(1, 2) ); // 3

А вот функция без аргументов, в этом случае достаточно указать только тело:

let sayHi = new Function('alert("Hello")');

sayHi(); // Hello

Главное отличие от других способов объявления функции, которые были рассмотрены ранее, заключается в том, что функция создаётся полностью «на лету» из строки, переданной во время выполнения.

Все предыдущие объявления требовали писать объявление функции в скрипте. Но new Function позволяет превратить любую строку в функцию. Например, можно получить новую функцию с сервера и затем выполнить ее:

let str = ... код, полученный с сервера динамически ...

let func = new Function(str);

func();

Это используется в очень специфических случаях, например, когда получаем код с сервера для динамической компиляции функции из шаблона, в сложных веб-приложениях.

Когда функция создаётся с использованием new Function, в её [[Environment]] записывается ссылка не на текущее лексическое окружение, а на глобальное. Поэтому такая функция не имеет доступа к внешним переменным, только к глобальным.

function getFunc() {

let value = "test";

let func = new Function('alert(value)');

return func;

}

getFunc()(); // ошибка: value не определено

Сравним это с обычным объявлением:

function getFunc() {

let value = "test";

let func = function() { alert(value); };

return func;

}

getFunc()(); // "test", из лексического окружения функции getFunc

Эта особенность new Function очень полезна на практике. Представьте, что нужно создать функцию из строки. Код этой функции неизвестен во время написания скрипта (вот поэтому не используем обычные функции), а будет определён только в процессе выполнения. Можно получить код с сервера или другого ресурса.

Новая функция должна взаимодействовать с основным скриптом. Если бы она имела доступ к внешним переменным это привело бы к проблеме. Проблема в том, что перед отправкой JavaScript-кода на реальные работающие проекты код сжимается с помощью *минификатора* – специальной программы, которая уменьшает размер кода, удаляя комментарии, лишние пробелы, и, что самое главное, локальным переменным даются укороченные имена. Например, если в функции объявляется переменная let userName, то минификатор изменяет её на let a (или другую букву, если она не занята), и изменяет её везде. Обычно так делать безопасно, потому что переменная является локальной и никто снаружи не имеет к ней достп. И внутри функции минификатор заменяет каждое её упоминание. Минификаторы анализируют структуру кода, и поэтому ничего не ломают.

Так что если бы new Function и имела доступ к внешним переменным, она не смогла бы найти переименованную userName. Кроме того, такой код был бы архитектурно хуже и более подвержен ошибкам. Чтобы передать что-то в функцию, созданную как new Function, можно использовать ее аргументы.

1. **Объект функции.**

Как известно, в JavaScript функция – это значение. Каждое значение в JavaScript имеет свой тип. В JavaScript, функции – это объекты.

Можно представить функцию как «объект, который может делать какое-то действие». Функции можно не только вызывать, но использовать их как обычные объекты: добавлять/удалять свойства, передавать их по ссылке и т.д.

[**Свойство name**](https://learn.javascript.ru/function-object#svoystvo-name)

Объект функции содержит несколько полезных свойств. Например, имя функции доступно как свойство name:

function sayHi() {

alert("Hi");

}

alert(sayHi.name); // sayHi

Корректное имя присваивается даже в случае, если функция создается без имени и тут же присваивается:

let sayHi = function() {

alert("Hi");

};

alert(sayHi.name); // sayHi

Это даже работает в случае присваивания значения по умолчанию:

function f(sayHi = function() {}) {

alert(sayHi.name); // sayHi

}

f();

В спецификации это называется «контекстное имя»: если функция не имеет name, то JavaScript пытается понять его из контекста.

Также имена имеют и методы объекта:

let user = {

sayHi() {

// ...

},

sayBye: function() {

// ...

}

}

alert(user.sayHi.name); // sayHi

alert(user.sayBye.name); // sayBye

В этом нет никакой магии. Бывает, что корректное имя определить невозможно. В таких случаях свойство name имеет пустое значение. Например:

let arr = [function() {}];

alert( arr[0].name ); // <пустая строка>

На практике такое бывает редко, обычно функции имеют name.

[**Свойство length**](https://learn.javascript.ru/function-object#svoystvo-length)

Ещё одно встроенное свойство length содержит количество параметров функции в её объявлении. Например:

function f1(a) {}

function f2(a, b) {}

function many(a, b, ...more) {}

alert(f1.length); // 1

alert(f2.length); // 2

alert(many.length); // 2

Как видно, троеточие, обозначающее «остальные параметры», здесь не учитываются.

Свойство length иногда используется для [интроспекций](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D1%80%D0%BE%D1%81%D0%BF%D0%B5%D0%BA%D1%86%D0%B8%D1%8F_(%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)) в функциях, которые работают с другими функциями. Например, в коде ниже функция ask принимает в качестве параметров вопрос question и произвольное количество функций-обработчиков ответа handler. Когда пользователь отвечает на вопрос, функция вызывает обработчики. Можно передать два типа обработчиков:

* Функцию без аргументов, которая будет вызываться только в случае положительного ответа.
* Функцию с аргументами, которая будет вызываться в обоих случаях и возвращать ответ.

Чтобы вызвать обработчик handler правильно, надо проверить свойство handler.length. Идея состоит в том, чтобы иметь простой синтаксис обработчика без аргументов для положительных ответов (наиболее распространённый случай), но также и возможность передавать универсальные обработчики:

function ask(question, ...handlers) {

let isYes = confirm(question);

for(let handler of handlers) {

if (handler.length == 0) {

if (isYes) handler();

} else {

handler(isYes);

}

}

}

ask("Вопрос?", () => alert('Вы ответили да'), result => alert(result));

[**Пользовательские свойства**](https://learn.javascript.ru/function-object#polzovatelskie-svoystva)

Также можно добавить свои собственные свойства. Например, свойство counter для отслеживания общего количества вызовов:

function sayHi() {

alert("Hi");

sayHi.counter++;

}

sayHi.counter = 0;

sayHi(); // Hi

sayHi(); // Hi

alert( `Вызвана ${sayHi.counter} раза` ); // Вызвана 2 раза

**Свойство не есть переменная**

Свойство функции, назначенное как sayHi.counter = 0, не объявляет локальную переменную counter внутри неё. Другими словами, свойство counter и переменная let counter – это две независимые вещи. Можно использовать функцию как объект, хранить в ней свойства, но они никак не влияют на её выполнение. Переменные – это не свойства функции и наоборот.

Иногда свойства функции могут использоваться вместо замыканий. Например, можно переписать функцию-счетчик из вопроса про [замыкание](https://learn.javascript.ru/closure), используя её свойство:

function makeCounter() {

function counter() {

return counter.count++;

};

counter.count = 0;

return counter;

}

let counter = makeCounter();

alert( counter() ); // 0

alert( counter() ); // 1

Свойство count теперь хранится прямо в функции, а не в её внешнем лексическом окружении. Основное отличие такого подхода от замыкания в том, что если значение count живет во внешней переменной, то она не доступна для внешнего кода. Изменить её могут только вложенные функции. А если оно присвоено как свойство функции, то можно его получить:

function makeCounter() {

function counter() {

return counter.count++;

};

counter.count = 0;

return counter;

}

let counter = makeCounter();

counter.count = 10;

alert( counter() ); // 10

Поэтому выбор реализации зависит от целей разработчика.

**[Named Function Expression](https://learn.javascript.ru/function-object" \l "named-function-expression)**

Named Function Expression или NFE – это термин для Function Expression, у которого есть имя:

let sayHi = function func(who) {

alert(`Hello, ${who}`);

};

Заметьте, что функция всё ещё задана как Function Expression. Добавление "func" после function не превращает объявление в Function Declaration, потому что оно все еще является частью выражения присваивания. Добавление такого имени ничего не ломает. Функция все еще доступна как sayHi():

let sayHi = function func(who) {

alert(`Hello, ${who}`);

};

sayHi("John"); // Hello, John

Есть две важные особенности имени func, ради которого оно даётся:

1. Оно позволяет функции ссылаться на себя же.
2. Оно не доступно за пределами функции.

Например, ниже функция sayHi вызывает себя с "Guest", если не передан параметр who:

let sayHi = function func(who) {

if (who) {

alert(`Hello, ${who}`);

} else {

func("Guest");

}

};

sayHi(); // Hello, Guest

func(); // Ошибка

Не следует использовать имя sayHi для вложенного вызова, так как значение sayHi может быть изменено. Функция может быть присвоена другой переменной, и тогда код начнет выдавать ошибки:

let sayHi = function(who) {

if (who) {

alert(`Hello, ${who}`);

} else {

sayHi("Guest"); // Ошибка

}

};

let welcome = sayHi;

sayHi = null;

welcome(); // Ошибка

Так происходит, потому что функция берет sayHi из внешнего лексического окружения. Так как локальная переменная sayHi отсутствует, используется внешняя. И на момент вызова эта внешняя sayHi равна null. Необязательное имя, которое можно вставить в Function Expression, как раз и призвано решать такого рода проблемы. Все работает, потому что имя "func" локальное и находится внутри функции. Теперь оно взято не снаружи (и недоступно оттуда). Спецификация гарантирует, что оно всегда будет ссылаться на текущую функцию. Внешний код все еще содержит переменные sayHi и welcome, но теперь func – это «внутреннее имя функции», таким образом она может вызвать себя изнутри.

Задать «внутреннее» имя можно только для Function Expression, и не нельзя для Function Declaration. Если нужно надёжное «внутреннее» имя, стоит переписать Function Declaration на Named Function Expression.

1. **Остаточные параметры и оператор расширения.**

Многие встроенные функции JavaScript поддерживают произвольное количество аргументов. Например: Math.max(arg1, arg2, ..., argN) – вычисляет максимальное число из переданных; Object.assign(dest, src1, ..., srcN) – копирует свойства из исходных объектов src1..N в целевой объект dest и др.

[**Остаточные параметры (...)**](https://learn.javascript.ru/rest-parameters-spread-operator#ostatochnye-parametry)

Вызывать функцию можно с любым количеством аргументов, независимо от того, как она была определена. Например:

function sum(a, b) {

return a + b;

}

alert( sum(1, 2, 3, 4, 5) );

Лишние аргументы не вызовут ошибку, но приняты будут только первые два.

Остаточные параметры могут быть обозначены через три точки «...». Суть его в том, что оставшиеся параметры помещаются в массив. Например, соберём все аргументы в массив args:

function sumAll(...args) {

let sum = 0;

for (let arg of args) sum += arg;

return sum;

}

alert( sumAll(1) ); // 1

alert( sumAll(1, 2) ); // 3

alert( sumAll(1, 2, 3) ); // 6

Можно положить первые несколько параметров в переменные и собрать в массив остальные. В примере ниже первые два аргумента функции станут именем и фамилией, а третий и последующие превратятся в массив titles:

function showName(firstName, lastName, ...titles) {

alert( firstName + ' ' + lastName ); // Юлий Цезарь

// titles = ["Консул", "Император"]

alert( titles[0] ); // Консул

alert( titles[1] ); // Император

alert( titles.length ); // 2

}

showName("Юлий", "Цезарь", "Консул", "Император");

Остаточные параметры собирают все остальные аргументы, поэтому бессмысленно писать что-либо после них. Это вызовет ошибку:

function f(arg1, ...rest, arg2) { // Ошибка

}

Все аргументы функции находятся в псевдомассиве arguments под своими порядковыми номерами. Например:

function showName() {

alert( arguments.length );

alert( arguments[0] );

alert( arguments[1] );

}

// Вывод: 2, Юлий, Цезарь

showName("Юлий", "Цезарь");

// Вывод: 1, Илья, undefined

showName("Илья");

Раньше в языке не было остаточных параметров, и получить все аргументы функции можно было только с помощью arguments. Этот способ всё ещё работает, его можно найти в старом коде. Но у него есть один недостаток. Хотя arguments похож на массив, и он тоже перебираемый, это всё же не массив. Он не поддерживает методы массивов, поэтому нельзя, например, вызвать arguments.map(...). К тому же, arguments всегда содержит все аргументы функции – нельзя получить их часть. А остаточные параметры позволяют это сделать.

Соответственно, для более удобной работы с аргументами лучше использовать остаточные параметры.

У стрелочных функций нет своего объекта arguments. Если обратиться к arguments из стрелочной функции, то получим аргументы внешней обычной функции. Пример:

function f() {

let showArg = () => alert(arguments[0]);

showArg(2);

}

f(1); // 1

**[Оператор расширения](https://learn.javascript.ru/rest-parameters-spread-operator" \l "spread-operator)**

Иногда нужно массив преобразовать в список параметров. Например, есть встроенная функция [Math.max](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/max). Она возвращает наибольшее число из списка:

alert( Math.max(3, 5, 1) ); // 5

Если вызвать эту функцию для массива чисел [3, 5, 1], то она его не обработает, так как ожидает список параметров:

let arr = [3, 5, 1];

alert( Math.max(arr) ); // NaN

Чтобы преобразовать массив в список необходимо использовать оператор расширения. Он похож на остаточные параметры – тоже использует ..., но делает совершенно противоположное. Когда ...arr используется при вызове функции, он «расширяет» перебираемый объект arr в список аргументов. Для Math.max:

let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5

Этим же способом можно передать несколько итерируемых объектов и комбинировать оператор расширения с обычными значениями:

let arr1 = [1, -2, 3, 4];

let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25

Оператор расширения можно использовать и для слияния массивов:

let arr = [3, 5, 1];

let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15

Оператора расширения работает с любым перебираемым объектом. Например, оператор расширения подойдёт для того, чтобы превратить строку в массив символов:

let str = "Привет";

alert( [...str] ); // П,р,и,в,е,т

Оператор расширения использует итераторы, чтобы собирать элементы. Так же, как это делает for..of. Цикл for..of перебирает строку как последовательность символов, поэтому из ...str получается "П", "р", "и", "в", "е", "т". Получившиеся символы собираются в массив при помощи стандартного объявления массива: [...str].

Для этой задачи можно использовать и Array.from. Он тоже преобразует перебираемый объект (такой как строка) в массив:

let str = "Привет";

alert( Array.from(str) ); // П,р,и,в,е,т

Результат аналогичен [...str]. Но между Array.from(obj) и [...obj] есть разница: Array.from работает как с псевдомассивами, так и с итерируемыми объектами; оператор расширения работает только с итерируемыми объектами. Таким образом, если нужно сделать из чего угодно массив, Array.from – более универсальный метод.

1. **Каррирование**

[*Каррирование*](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%80%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) или карринг (currying) в функциональном программирование – это преобразование функции с множеством аргументов в набор вложенных функций с одним аргументом. При вызове каррированной функции с передачей ей одного аргумента, она возвращает новую функцию, которая ожидает поступления следующего аргумента. Новые функции, ожидающие следующего аргумента, возвращаются при каждом вызове каррированной функции – до тех пор, пока функция не получит все необходимые ей аргументы. Ранее полученные аргументы, благодаря механизму замыканий, ждут того момента, когда функция получит всё, что ей нужно для выполнения вычислений. После получения последнего аргумента функция выполняет вычисления и возвращает результат.

Говоря о [каррировании](https://medium.com/@kbrainwave/currying-in-javascript-ce6da2d324fe), можно сказать, что это процесс превращения функции с несколькими аргументами в функцию с меньшей арностью.

[*Арность*](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D0%BD%D0%BE%D1%81%D1%82%D1%8C) – это количество аргументов функции. Например – вот объявление пары функций:

function fn(a, b) {

*//...*

}

function \_fn(a, b, c) {

*//...*

}

Функция fn принимает два аргумента (это бинарная или 2-арная функция), функция \_fn принимает три аргумента (тернарная, 3-арная функция).

В результате каррирования, функция с несколькими аргументами преобразуется в набор функций, каждая из которых принимает один аргумент. Рассмотрим пример:

function multiply(a, b, c) {

   return a \* b \* c;

}

Функция multiply принимает три аргумента и возвращает их произведение:  
  
multiply(1,2,3); *// 6*

Преобразовать её к набору функций, каждая из которых принимает один аргумент. Создадим каррированный вариант этой функции и посмотрим на то, как получить тот же результат в ходе вызова нескольких функций:

function multiply(a) {

   return (b) => {

       return (c) => {

           return a \* b \* c

       }

   }

}

log(multiply(1)(2)(3)) *// 6*

В примере вызов единственной функции с тремя аргументами multiply(1,2,3) преобразован к вызову трёх функций – multiply(1)(2)(3). При использовании новой конструкции каждая функция, кроме последней, возвращающей результат вычислений, принимает аргумент и возвращает другую функцию, также способную принять аргумент и возвратить другую функцию. Распишем конструкцию вида multiply(1)(2)(3), чтобы она была более понятной:

const mul1 = multiply(1)*;*

const mul2 = mul1(2)*;*

const result = mul2(3)*;*

log(result)*; // 6*

Подробнее разберём, что здесь происходит. Сначала передается аргумент 1 функции multiply:

const mul1 = multiply(1);

При работе этой функции срабатывает такая конструкция:

return (b) => {

       return (c) => {

           return a \* b \* c

       }

   }

Теперь в mul1 имеется ссылка на функцию, принимающую аргумент b. Вызовем функцию mul1, передав ей 2:

const mul2 = mul1(2);

В результате этого вызова выполнится следующий код:

return (c) => {

           return a \* b \* c

       }

Константа mul2 будет содержать ссылку на функцию, которая могла бы оказаться в ней, например, в результате выполнения следующей операции:

mul2 = (c) => {

           return a \* b \* c

       }

Если теперь вызвать функцию mul2, передав ей 3, то функция выполнит необходимые вычисления, воспользовавшись аргументами a и b:

const result = mul2(3);

Результатом этих вычислений будет 6:

log(result)*; // 6*

Функция mul2, обладающая самым большим уровнем вложенности, имеет доступ к областям видимости, к замыканиям, формируемым функциями multiply и mul1. Именно поэтому в функции mul2можно производить вычисления с переменными, объявленными в функциях, выполнение которых уже завершено, которые уже возвратили некие значения и обработаны сборщиком мусора.

**Каррирование и частичное применение функций**

Сейчас, возникает ощущение, что количество вложенных функций, при представлении функции в виде набора вложенных функций, зависит от количества аргументов функции. И, если речь идёт о каррировании, то это так. Особый вариант функции для вычисления объёма, можно сделать таким:

function volume(l) {

   return (w, h) => {

       return l \* w \* h

   }

}

Здесь применены идеи, очень похожие на рассмотренные выше. Пользоваться этой функцией можно так:

const hV = volume(70)*;*

hV(203,142)*;*

hV(220,122)*;*

hV(120,123)*;*

А можно и так:

volume(70)(90,30)*;*

volume(70)(390,320)*;*

volume(70)(940,340)*;*

Фактически, здесь командой volume(70), создана специализированная функция для вычисления объёма тел, одно из измерений которых (а именно – длина, l), зафиксировано. Функция volume ожидает 3 аргумента и содержит 2 вложенных функции, в отличие от предыдущей версии подобной функции, каррированный вариант которой содержал 3 вложенных функции.

Та функция, которая получилась после вызова volume(70) реализует концепцию частичного применения функции (partial function application). Каррирование и частичное применение функций очень похожи друг на друга, но концепции это разные.

При частичном применении функцию преобразуют в другую функцию, обладающую меньшим числом аргументов (меньшей арностью). Некоторые аргументы такой функции оказываются зафиксированными (для них задаются значения по умолчанию).

Каррирование и частичное применение функций может оказаться полезным в различных ситуациях. Например – при разработке небольших модулей, подходящих для повторного использования.

Частичное применение функций позволяет облегчить использование универсальных модулей. Например, есть интернет-магазин, в коде которого имеется функция, которая используется для вычисления суммы к оплате с учётом скидки.

function discount(price, discount) {

   return price \* discount

}

Есть определённая категория клиентов, которые получают скидку в 10%. Например, если такой клиент покупает что-то на $500, то он получает скидку размером $50:

const price = discount(500,0.10); // $50

// $500 - $50 = $450

Несложно заметить, что при таком подходе, постоянно придётся вызывать эту функцию с двумя аргументами:

const price = discount(1500,0.10); // $150

// $1,500 - $150 = $1,350

const price = discount(2000,0.10); // $200

// $2,000 - $200 = $1,800

const price = discount(50,0.10); // $5

// $50 - $5 = $45

const price = discount(5000,0.10); // $500

// $5,000 - $500 = $4,500

const price = discount(300,0.10); // $30

// $300 - $30 = $270

Исходную функцию можно привести к такому виду, который позволял бы получать новые функции с заранее заданным уровнем скидки, при вызове которых им достаточно передавать сумму покупки. Функция discount() в примере имеет два аргумента. Преобразуем ее следующим образом:

function discount(discount) {

   return (price) => {

       return price \* discount;

   }

}

const tenPercentDiscount = discount(0.1);

Функция tenPercentDiscount() представляет собой результат частичного применения функции discount(). При вызове tenPercentDiscount() этой функции достаточно передать цену, а скидка в 10%, то есть – аргумент discount, уже будет задана:

tenPercentDiscount(500); // $50

// $500 - $50 = $450

Если в магазине имеются покупатели, которым решено дать скидку размером в 20%, то получить соответствующую функцию для работы с ними можно так:

const twentyPercentDiscount = discount(0.2);

Теперь функцию twentyPercentDiscount() можно вызывать для расчёта стоимости товаров с учётом скидки в 20%:

twentyPercentDiscount(500); // 100

// $500 - $100 = $400

twentyPercentDiscount(5000); // 1000

// $5,000 - $1,000 = $4,000

twentyPercentDiscount(1000000); // 200000

// $1,000,000 - $200,000 = $600,000

1. **Генераторы**

Обычные функции возвращают только одно-единственное значение (или ничего). Генераторы могут порождать (yield) множество значений одно за другим, по мере необходимости. Генераторы отлично работают с перебираемыми объектами и позволяют легко создавать потоки данных.

**[Функция-генератор](https://learn.javascript.ru/generators" \l "funktsiya-generator)**

Для объявления генератора используется специальная синтаксическая конструкция: function\* f(…) (или function \*f(…)), которая называется «функция-генератор». Выглядит она так:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

Функции-генераторы ведут себя не так, как обычные. Когда такая функция вызвана, она не выполняет свой код. Вместо этого она возвращает специальный объект, так называемый «генератор» для управления её выполнением:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

let generator = generateSequence();

alert(generator); // [object Generator]

Основным методом генератора является next(). При вызове он запускает выполнение кода до ближайшей инструкции yield <значение> (значение может отсутствовать, в этом случае оно полагается равным undefined). По достижении yield выполнение функции приостанавливается, а соответствующее значение – возвращается во внешний код:

Результатом метода next() всегда является объект с двумя свойствами:

* value: значение из yield.
* done: true, если выполнение функции завершено, иначе false.

Создадим генератор и получим первое из возвращаемых им значений:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

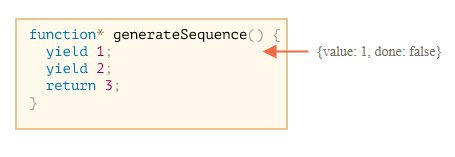
}

let generator = generateSequence();

let one = generator.next();

alert(JSON.stringify(one)); // {value: 1, done: false}

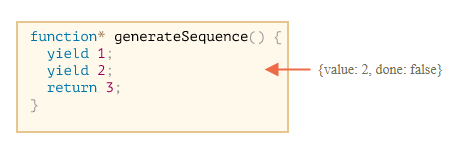
На данный момент получено только первое значение, выполнение функции остановлено на второй строке:



Повторный вызов generator.next() возобновит выполнение кода и вернёт результат следующего yield:

let two = generator.next();

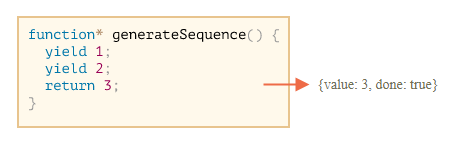
alert(JSON.stringify(two)); // {value: 2, done: false}



И, наконец, последний вызов завершит выполнение функции и вернёт результат return:

let three = generator.next();

alert(JSON.stringify(three)); // {value: 3, done: true}



Сейчас генератор полностью выполнен. Можно увидеть это по свойству done:true и обработать value:3 как окончательный результат. Новые вызовы generator.next() больше не имеют смысла. Впрочем, если они и будут, то не вызовут ошибки, но будут возвращать один и тот же объект: {done: true}.

[**Перебор генераторов**](https://learn.javascript.ru/generators#perebor-generatorov)

Генераторы являются [перебираемыми](https://learn.javascript.ru/iterable) объектами. Возвращаемые ими значения можно перебирать через for..of:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

let generator = generateSequence();

for(let value of generator) {

alert(value); // 1, затем 2

}

Обратите внимание: пример выше выводит значение 1, затем 2. Значение 3 выведено не будет. Это потому что перебор через for..of игнорирует последнее значение, при котором done: true. Поэтому, если необходимо, чтобы были все значения при переборе через for..of, то надо возвращать их через yield:

function\* generateSequence() {

yield 1;

yield 2;

yield 3;

}

let generator = generateSequence();

for(let value of generator) {

alert(value); // 1, затем 2, затем 3

}

Так как генераторы являются перебираемыми объектами, можно использовать всю связанную с ними функциональность, например, оператор расширения ...:

function\* generateSequence() {

yield 1;

yield 2;

yield 3;

}

let sequence = [0, ...generateSequence()];

alert(sequence); // 0, 1, 2, 3

В коде выше ...generateSequence() превращает перебираемый объект-генератор в массив элементов.

В примерах выше генерируются конечные последовательности, но также можно сделать генератор, который будет возвращать значения бесконечно. Например, бесконечная последовательность псевдослучайных чисел. Тут потребуется break (или return) в цикле for..of по такому генератору, иначе цикл будет продолжаться бесконечно и скрипт «зависнет».

[**Композиция генераторов**](https://learn.javascript.ru/generators#kompozitsiya-generatorov)

Композиция генераторов – это особенная возможность генераторов, которая позволяет прозрачно «встраивать» генераторы друг в друга. Например, есть функция для генерации последовательности чисел:

function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) yield i;

}

Надо использовать её при генерации более сложной последовательности: сначала цифры 0..9 (с кодами символов 48…57), за которыми следуют буквы алфавита a..z (коды символов 65…90), за которыми следуют буквы в верхнем регистре A..Z (коды символов 97…122). Можно использовать такую последовательность для генерации паролей, выбирать символы из неё (может быть, ещё добавить символы пунктуации), но сначала её нужно сгенерировать.

В обычной функции, чтобы объединить результаты из нескольких других функций, надо вызвать их, сохранить промежуточные результаты, а затем в конце их объединить. Для генераторов есть особый синтаксис yield\*, который позволяет «вкладывать» генераторы один в другой (осуществлять их композицию). Пример генератора с композицией:

function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) yield i;

}

function\* generatePasswordCodes() {

// 0..9

yield\* generateSequence(48, 57);

// A..Z

yield\* generateSequence(65, 90);

// a..z

yield\* generateSequence(97, 122);

}

let str = '';

for(let code of generatePasswordCodes()) {

str += String.fromCharCode(code);

}

alert(str); // 0..9A..Za..z

Директива yield\* делегирует выполнение другому генератору. Этот термин означает, что yield\* gen перебирает генератор gen и прозрачно направляет его вывод наружу. Как если бы значения были сгенерированы внешним генератором.

Результат – такой же, как если встроить код из вложенных генераторов:

function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) yield i;

}

function\* generateAlphaNum() {

// yield\* generateSequence(48, 57);

for (let i = 48; i <= 57; i++) yield i;

// yield\* generateSequence(65, 90);

for (let i = 65; i <= 90; i++) yield i;

// yield\* generateSequence(97, 122);

for (let i = 97; i <= 122; i++) yield i;

}

let str = '';

for(let code of generateAlphaNum()) {

str += String.fromCharCode(code);

}

alert(str); // 0..9a..zA..Z

Композиция генераторов – естественный способ вставлять вывод одного генератора в поток другого. Она не использует дополнительную память для хранения промежуточных результатов.

Директива yield не только возвращает результат наружу, но и может передавать значение извне в генератор. Чтобы это сделать, нужно вызвать generator.next(arg) с аргументом. Этот аргумент становится результатом yield. Продемонстрируем это на примере:

function\* gen() {

// Передаём вопрос во внешний код и ожидаем ответа

let result = yield "2 + 2 = ?"; // (\*)

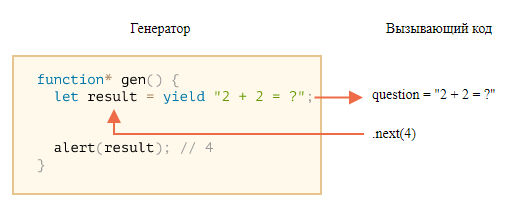
alert(result);

}

let generator = gen();

let question = generator.next().value; // <-- yield возвращает значение

generator.next(4); // --> передаём результат в генератор



1. Первый вызов generator.next() – всегда без аргумента, он начинает выполнение и возвращает результат первого yield "2+2=?". На этой точке генератор приостанавливает выполнение.
2. Затем, как показано на картинке выше, результат yield переходит во внешний код в переменную question.
3. При generator.next(4) выполнение генератора возобновляется, а 4 выходит из присваивания как результат: let result = 4.

Обратите внимание, что внешний код не обязан немедленно вызывать next(4). Ему может потребоваться время, генератор подождёт. Например:

// возобновить генератор через некоторое время

setTimeout(() => generator.next(4), 1000);

Как видно, в отличие от обычных функций, генератор может обмениваться результатами с вызывающим кодом, передавая значения в next/yield. Чтобы сделать происходящее более очевидным, вот ещё один пример с большим количеством вызовов:

function\* gen() {

let ask1 = yield "2 + 2 = ?";

alert(ask1); // 4

let ask2 = yield "3 \* 3 = ?"

alert(ask2); // 9

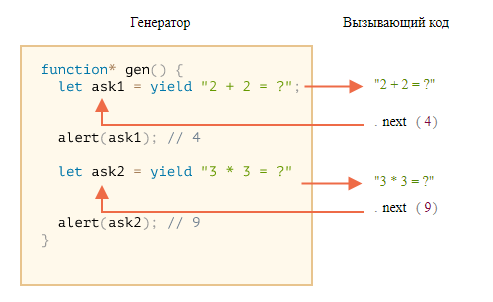
}

let generator = gen();

alert( generator.next().value ); // "2 + 2 = ?"

alert( generator.next(4).value ); // "3 \* 3 = ?"

alert( generator.next(9).done ); // true



1. Первый .next() начинает выполнение, доходит до первого yield.
2. Результат возвращается во внешний код.
3. Второй .next(4) передаёт 4 обратно в генератор как результат первого yield и возобновляет выполнение.
4. Выполнение доходит до второго yield, который станет результатом .next(4).
5. Третий next(9) передаёт 9 в генератор как результат второго yield и возобновляет выполнение, которое завершается окончанием функции, так что done: true.

Таким образом, каждый next(value) передаёт в генератор значение, которое становится результатом текущего yield, возобновляет выполнение и получает выражение из следующего yield.

[**generator.throw**](https://learn.javascript.ru/generators#generator-throw)

Внешний код может передавать значение в генератор как результат yield. Но можно передать не только результат, но и инициировать (throw) ошибку. Это естественно, так как ошибка является своего рода результатом. Для того, чтобы передать ошибку в yield, нужно вызвать generator.throw(err). В этом случае, исключение err возникнет на строке с yield. Например, здесь yield "2 + 2 = ?" приведёт к ошибке:

function\* gen() {

try {

let result = yield "2 + 2 = ?"; // (1)

alert("Выполнение программы не дойдет до этой строки, потому что выше возникнет исключение");

} catch(e) {

alert(e); // покажет ошибку

}

}

let generator = gen();

let question = generator.next().value;

generator.throw(new Error("Ответ не найден в моей базе данных")); // (2)

Ошибка, которая проброшена в генератор на строке (2), приводит к исключению на строке (1) с yield. В примере выше try..catch перехватывает её и отображает. Если необходимо перехватывать её, то она, как и любое обычное исключение, будет передана из генератора во внешний код. Текущая строка вызывающего кода – это строка с generator.throw, отмечена (2). Таким образом, можно отловить её во внешнем коде, как здесь:

function\* generate() {

let result = yield "2 + 2 = ?"; // Ошибка в этой строке

}

let generator = generate();

let question = generator.next().value;

try {

generator.throw(new Error("Ответ не найден в моей базе данных"));

} catch(e) {

alert(e); // покажет ошибку

}

1. **Колбэки.**

Многие действия в JavaScript асинхронные. Например, рассмотрим функцию loadScript(src):

function loadScript(src) {

let script = document.createElement('script');

script.src = src;

document.head.append(script);

}

Эта функция загружает на страницу новый скрипт. Когда в тело документа добавится конструкция <script src="…">, браузер загрузит скрипт и выполнит его. Пример использования этой функции:

loadScript('/my/script.js');

Такие функции называют «асинхронными», потому что действие (загрузка скрипта) будет завершено не сейчас, а потом. Если после вызова loadScript(…) есть какой-то код, то он не будет ждать, пока скрипт загрузится.

loadScript('/my/script.js');

Доустпим надо использовать новый скрипт, как только он будет загружен. Например, он объявляет новую функцию, которую надо выполнить. Если просто вызовать эту функцию после loadScript(…), то ничего не выйдет:

loadScript('/my/script.js');

newFunction(); // такой функции не существует

Действительно, ведь у браузера не было времени загрузить скрипт. Сейчас функция loadScript никак не позволяет отследить момент загрузки. Скрипт загружается, а потом выполняется. Но нужно точно знать, когда это произойдет, чтобы использовать функции и переменные из этого скрипта. Давайте передадим функцию callback вторым аргументом в loadScript, чтобы вызвать ее, когда скрипт загрузится:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(script);

document.head.append(script);

}

Теперь, если надо вызвать функцию из скрипта, нужно делать это в колбэке:

loadScript('/my/script.js', function() {

newFunction();

...

});

Суть заключается в том, что вторым аргументом передается функция (обычно анонимная), которая выполняется по завершению действия. Возьмём для примера реальный скрипт с библиотекой функций:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(script);

document.head.append(script);

}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {

alert(`Здорово, скрипт ${script.src} загрузился`);

alert( \_ );

});

Такое написание называют асинхронным программированием с использованием колбэков. В функции, которые выполняют какие-либо асинхронные операции, передается аргумент callback – функция, которая будет вызвана по завершению асинхронного действия.

[**Колбэк в колбэке**](https://learn.javascript.ru/callbacks#kolbek-v-kolbeke)

Допустим, надо загрузить два скрипта один за другим: сначала первый, а за ним второй. Можно вызвать loadScript еще раз уже внутри колбэка:

loadScript('/my/script.js', function(script) {

alert(`Здорово, скрипт ${script.src} загрузился, загрузим еще один`);

loadScript('/my/script2.js', function(script) {

alert(`Здорово, второй скрипт загрузился`);

});

});

Когда внешняя функция loadScript выполнится, вызовется та, что внутри колбэка. Загрузим еще один скрипт:

loadScript('/my/script.js', function(script) {

loadScript('/my/script2.js', function(script) {

loadScript('/my/script3.js', function(script) {

// ...и так далее, пока все скрипты не будут загружены

});

})

});

Возникает необходимость каждое новое действие вызывать внутри колбэка. Этот вариант подойдет, когда необходимо одно-два действия, но для большего количества уже не удобно. Альтернативные подходы разберём позже.

**[Перехват ошибок](https://learn.javascript.ru/callbacks" \l "perehvat-oshibok)**

Допустим, что загрузить скрипт не удалось, тогда колбэк должен уметь реагировать на возможные проблемы. Ниже улучшенная версия loadScript, которая умеет отслеживать ошибки загрузки:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(null, script);

script.onerror = () => callback(new Error(`Не удалось загрузить скрипт ${src}`));

document.head.append(script);

}

Вызывается callback(null, script) в случае успешной загрузки и callback(error), если загрузить скрипт не удалось. Например:

loadScript('/my/script.js', function(error, script) {

if (error) {

// обрабатываем ошибку

} else {

// скрипт успешно загружен

}

});

Опять же, подход, который был использован в loadScript, также распространен и называется «колбэк с первым аргументом-ошибкой» («error-first callback»). Правила таковы:

1. Первый аргумент функции callback зарезервирован для ошибки. В этом случае вызов выглядит вот так: callback(err).
2. Второй и последующие аргументы — для результатов выполнения. В этом случае вызов выглядит вот так: callback(null, result1, result2…).

Одна и та же функция callback используется и для информирования об ошибке, и для передачи результатов.

[**Адская пирамида вызовов**](https://learn.javascript.ru/callbacks#adskaya-piramida-vyzovov)

На первый взгляд это рабочий способ написания асинхронного кода. Так и есть. Для одного или двух вложенных вызовов всё выглядит нормально. Но для нескольких асинхронных действий, которые нужно выполнить друг за другом, код выглядит вот так:

loadScript('1.js', function(error, script) {

if (error) {

handleError(error);

} else {

// ...

loadScript('2.js', function(error, script) {

if (error) {

handleError(error);

} else {

// ...

loadScript('3.js', function(error, script) {

if (error) {

handleError(error);

} else {

// ...и так далее, пока все скрипты не будут загружены (\*)

}

});

}

})

}

});

В примере выше:

1. Загружаем 1.js. Продолжаем, если нет ошибок.
2. Загружаем 2.js. Продолжаем, если нет ошибок.
3. Загружаем 3.js. Продолжаем, если нет ошибок. И так далее (\*).

Чем больше вложенных вызовов, тем код будет иметь всё большую вложенность, которую сложно поддерживать. Такой подход к написанию кода не приветствуется. Один из лучших способов – использовать промисы.

1. **Промис (Promise)**

Рассмотрим ситуацию, которая часто встречается в программировании. Есть «создающий» код, который делает что-то, что занимает время. Например, загружает данные по сети. Есть «потребляющий» код, который хочет получить результат «создающего» кода, когда он будет готов. Он может быть необходим более чем одной функции.

*Promise* (по англ. promise) – это специальный объект в JavaScript, который связывает «создающий» и «потребляющий» коды вместе. «Создающий» код может выполняться сколько потребуется, чтобы получить результат, а промис делает результат доступным для кода, который подписан на него, когда результат готов.

Синтаксис создания Promise:

let promise = new Promise(function(resolve, reject) {

});

Функция, переданная в конструкцию new Promise, называется исполнитель (executor). Когда Promise создаётся, она запускается автоматически. Она должна содержать «создающий» код, который когда-нибудь создаст результат. Её аргументы resolve и reject – это колбэки, которые предоставляет сам JavaScript. Наш код – только внутри исполнителя.

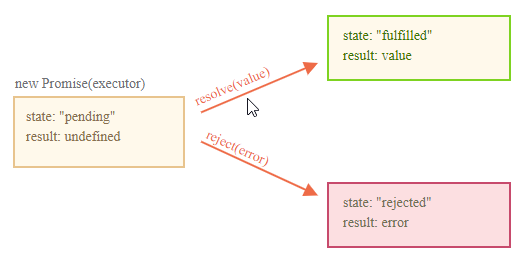
Когда он получает результат, сейчас или позже, он должен вызвать один из этих колбэков:

* resolve(value) – если работа завершилась успешно, с результатом value.
* reject(error) – если произошла ошибка, error – объект ошибки.

У объекта promise, возвращаемого конструктором new Promise, есть внутренние свойства:

* state («состояние») – вначале "pending" («ожидание»), потом меняется на "fulfilled" («выполнено успешно») при вызове resolve или на "rejected" («выполнено с ошибкой») при вызове reject.
* result («результат») – вначале undefined, далее изменяется на value при вызове resolve(value) или на error при вызове reject(error).

Так что исполнитель по итогу переводит promise одно из двух состояний:



Ниже пример конструктора Promise и простого исполнителя с кодом, дающим результат с задержкой (через setTimeout):

let promise = new Promise(function(resolve, reject) {

// функция выполнится автоматически, при вызове new Promise

// через 1 секунду сигнализировать, что задача выполнена с результатом "done"

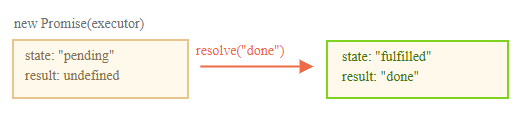
setTimeout(() => resolve("done"), 1000);

});

Можно наблюдать две вещи, запустив код выше:

1. Функция-исполнитель запускается сразу же при вызове new Promise.
2. Исполнитель получает два аргумента: resolve и reject – это функции, встроенные в JavaScript, поэтому разработчику не нужно их писать. Ему нужно лишь позаботиться, чтобы исполнитель вызвал одну из них по готовности.

Спустя одну секунду «обработки» исполнитель вызовет resolve("done"), чтобы передать результат:



Это был пример успешно выполненной задачи, в результате получен «успешно выполненный» промис.

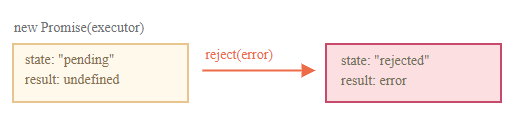
Рассмотрим пример, в котором исполнитель сообщит, что задача выполнена с ошибкой:

let promise = new Promise(function(resolve, reject) {

// спустя одну секунду будет сообщено, что задача выполнена с ошибкой

setTimeout(() => reject(new Error("Whoops!")), 1000);

});



Промис – и успешный, и отклонённый будем называть «завершённым», в отличие изначального промиса «в ожидании». Исполнитель должен вызвать что-то одно: resolve или reject. Состояние промиса может быть изменено только один раз. Все последующие вызовы resolve и reject будут проигнорированы:

let promise = new Promise(function(resolve, reject) {

resolve("done");

reject(new Error("…")); // игнорируется

setTimeout(() => resolve("…")); // игнорируется

});

Также заметим, что функция resolve/reject ожидает только один аргумент (или ни одного). Все дополнительные аргументы будут проигнорированы.

В случае, если что-то пошло не так, необходимо вызвать reject. Это можно сделать с аргументом любого типа (как и resolve), но рекомендуется использовать объект Error (или унаследованный от него).

Обычно исполнитель делает что-то асинхронное и после этого вызывает resolve/reject, то есть через какое-то время. Но это не обязательно, resolve или reject могут быть вызваны сразу:

let promise = new Promise(function(resolve, reject) {

// задача, не требующая времени

resolve(123); // мгновенно выдаст результат: 123

});

Свойства state и result – это внутренние свойства объекта Promise и к ним нет прямого доступа. Для обработки результата следует использовать методы .then/.catch/.finally.

Объект Promise служит связующим звеном между исполнителем («создающим» кодом) и функциями-потребителями, которые получат либо результат, либо ошибку. Функции-потребители могут быть зарегистрированы (подписаны) с помощью методов .then, .catch и .finally.

[**then**](https://learn.javascript.ru/promise-basics#then)

Наиболее важный и фундаментальный метод – .then. Синтаксис:

promise.then(

function(result) { /\* обработает успешное выполнение \*/ },

function(error) { /\* обработает ошибку \*/ }

);

Первый аргумент метода .then – функция, которая выполняется, когда промис переходит в состояние «выполнен успешно», и получает результат. Второй аргумент .then – функция, которая выполняется, когда промис переходит в состояние «выполнен с ошибкой», и получает ошибку. Например, реакция на успешно выполненный промис:

let promise = new Promise(function(resolve, reject) {

setTimeout(() => resolve("done!"), 1000);

});

promise.then(

result => alert(result), // выведет "done!" через одну секунду

error => alert(error) // не будет запущена

);

Выполнилась первая функция. А в случае ошибки в промисе – выполнится вторая:

let promise = new Promise(function(resolve, reject) {

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

promise.then(

result => alert(result), // не будет запущена

error => alert(error) // выведет "Error: Whoops!" спустя одну секунду

);

Если интересует только результат успешного выполнения задачи, то в then можно передать только одну функцию:

let promise = new Promise(resolve => {

setTimeout(() => resolve("done!"), 1000);

});

promise.then(alert); // выведет "done!" спустя одну секунду

[**catch**](https://learn.javascript.ru/promise-basics#catch)

Если надо обработать ошибку, то можно использовать null в качестве первого аргумента: .then(null, errorHandlingFunction). Или можно воспользоваться методом .catch(errorHandlingFunction), который сделает тоже самое:

let promise = new Promise((resolve, reject) => {

setTimeout(() => reject(new Error("Ошибка!")), 1000);

});

promise.catch(alert); // выведет "Error: Ошибка!" спустя одну секунду

Вызов .catch(f) – это сокращенный, «укороченный» вариант .then(null, f).

**[finally](https://learn.javascript.ru/promise-basics" \l "finally)**

По аналогии с блоком finally из обычного try {...} catch {...}, у промисов также есть метод finally. Вызов .finally(f) похож на .then(f, f), в том смысле, что f выполнится в любом случае, когда промис завершится: успешно или с ошибкой. Например:

new Promise((resolve, reject) => {

/\* сделать что-то, что займет время, и после вызвать resolve/reject \*/

})

// выполнится, когда промис завершится, независимо от того, успешно или нет

.finally(() => остановить индикатор загрузки)

.then(result => показать результат, err => показать ошибку)

Но это не совсем псевдоним then(f, f), как можно было подумать. Существует несколько важных отличий:

1. Обработчик, вызываемый из finally, не имеет аргументов. В finally не известно, как был завершён промис. И это нормально, потому что обычно задача состоит в том, чтобы выполнить «общие» завершающие процедуры.
2. Обработчик finally «пропускает» результат или ошибку дальше, к последующим обработчикам.

Например, здесь результат проходит через finally к then:

new Promise((resolve, reject) => {

setTimeout(() => resolve("result"), 2000)

})

.finally(() => alert("Промис завершён"))

.then(result => alert(result)); // <-- .then обработает результат

А здесь ошибка из промиса проходит через finally к catch:

new Promise((resolve, reject) => {

throw new Error("error");

})

.finally(() => alert("Промис завершён"))

.catch(err => alert(err)); // <-- .catch обработает объект ошибки

Это очень удобно, потому что finaly не предназначен для обработки результата промиса. Так что он просто пропускает его через себя дальше.

1. Последнее, но не менее значимое: вызов .finally(f) удобнее, чем.then(f, f) – не надо дублировать функции f.

Если промис в состоянии ожидания, обработчики в .then/catch/finally будут ждать его. Однако, если промис уже завершён, то обработчики выполнятся сразу:

// мгновенный перевод промиса в состояние "успешно завершён"

let promise = new Promise(resolve => resolve("готово!"));

promise.then(alert); // готово!

Рассмотрим несколько практических примеров того, как промисы могут облегчить написание асинхронного кода.

Ранее рассматривалась функция loadScript для загрузки скрипта. Реализация этой функции с использованием колбэков выглядит так:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(null, script);

script.onerror = () => callback(new Error(`Ошибка загрузки скрипта ${src}`));

document.head.append(script);

}

Перепишем её, используя Promise. Новой функции loadScript будет не нужен аргумент callback. Вместо этого она будет создавать и возвращать объект Promise, который будет переходить в состояние «успешно завершён», когда загрузка закончится. Внешний код может добавлять обработчики («подписчиков»), используя .then:

function loadScript(src) {

return new Promise(function(resolve, reject) {

let script = document.createElement('script');

script.src = src;

script.onload = () => resolve(script);

script.onerror = () => reject(new Error(`Ошибка загрузки скрипта ${src}`));

document.head.append(script);

});

}

Применение:

let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js");

promise.then(

script => alert(`${script.src} загружен!`),

error => alert(`Ошибка: ${error.message}`)

);

promise.then(script => alert('Ещё один обработчик...'));

Сразу заметно несколько преимуществ перед подходом с использованием колбэков:

| **Промисы** | **Колбэки** |
| --- | --- |
| Промисы позволяют выполнять действия в естественном порядке: сперва запустить loadScript(script), и затем (.then) выполнить действия с результатом. | Должна быть функция callback на момент вызова loadScript(script, callback). Другими словами, нужно знать что делать с результатом до того, как вызовется loadScript. |
| Можно вызывать .then у Promise столько раз, сколько надо, каждый раз добавляя новую функцию-подписчика. | Колбэк может быть только один. |

1. **Промисификация**

*Промисификация* – подразумевает преобразование функции, которая принимает колбэк таким образом, чтобы она вместо этого возвращала промис.

Такие преобразования часто необходимы в реальной жизни, так как многие функции и библиотеки основаны на колбэках, а использование промисов более удобно, поэтому есть смысл «промисифицировать» их. Например, ранее рассматриваемая функция loadScript(src, callback):

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(null, script);

script.onerror = () => callback(new Error(`Ошибка загрузки скрипта ${src}`));

document.head.append(script);

}

// использование:

// loadScript('path/script.js', (err, script) => {...})

Промисифицируем её. Новая функция loadScriptPromise(src) будет делать то же самое, но будет принимать только src (не callback) и возвращать промис.

let loadScriptPromise = function(src) {

return new Promise((resolve, reject) => {

loadScript(src, (err, script) => {

if (err) reject(err)

else resolve(script);

});

})

}

// использование:

// loadScriptPromise('path/script.js').then(...)

Теперь loadScriptPromise хорошо вписывается в код, основанный на промисах. Как видно, она передаёт всю работу исходной функции loadScript, предоставляя ей колбэк, по вызову которого происходит resolve/reject промиса.

На практике, как правило, есть необходимость промисифицировать не одну функцию, поэтому имеет смысл сделать для этого специальную «функцию-помощник» – promisify(f). Она будет принимать функцию для промисификации f и возвращать функцию-обёртку. Эта функция-обёртка делает то же самое, что и код выше: возвращает промис и передаёт вызов оригинальной f, отслеживая результат в своём колбэке:

function promisify(f) {

return function (...args) {

return new Promise((resolve, reject) => {

function callback(err, result) {

if (err) {

return reject(err);

} else {

resolve(result);

}

}

args.push(callback);

f.call(this, ...args);

});

};

};

// использование:

let loadScriptPromise = promisify(loadScript);

loadScriptPromise(...).then(...);

Здесь предполагается, что исходная функция ожидает колбэк с двумя аргументами (err, result). Но возможна ситуация, когда исходная f ожидает колбэк с большим количеством аргументов callback(err, res1, res2, ...). Ниже описана улучшенная функция promisify: при вызове promisify(f, true) результатом промиса будет массив результатов [res1, res2, ...]:

function promisify(f, manyArgs = false) {

return function (...args) {

return new Promise((resolve, reject) => {

function callback(err, ...results) {

if (err) {

return reject(err);

} else {

resolve(manyArgs ? results : results[0]);

}

}

args.push(callback);

f.call(this, ...args);

});

};

};

// использование:

f = promisify(f, true);

f(...).then(arrayOfResults => ..., err => ...)

Помните, промис может иметь только один результат, но колбэк технически может вызываться сколько угодно раз. Поэтому промисификация используется для функций, которые вызывают колбэк только один раз. Последующие вызовы колбэка будут проигнорированы.

1. **Async/await**

Существует специальный синтаксис для работы с промисами, который называется «async/await». Он прост для понимания и использования.

Ключевое слово *async* ставится перед функцией:

async function f() {

return 1;

}

У слова async один простой смысл: эта функция всегда возвращает промис. Значения других типов оборачиваются в завершившийся успешно промис автоматически. Например, эта функция возвратит выполненый промис с результатом 1:

async function f() {

return 1;

}

f().then(alert); // 1

Можно и явно вернуть промис, результат будет одинаковым:

async function f() {

return Promise.resolve(1);

}

f().then(alert); // 1

Так что ключевое слово async перед функцией гарантирует, что эта функция в любом случае вернёт промис.

Ключевое слово *await* можно использовать только внутри async-функций. Синтаксис:

// работает только внутри async–функций

let value = await promise;

Ключевое слово await заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от await не выполнится. После чего оно вернёт его результат, и выполнение кода продолжится. В этом примере промис успешно выполнится через 1 секунду:

async function f() {

let promise = new Promise((resolve, reject) => {

setTimeout(() => resolve("готово!"), 1000)

});

let result = await promise; // (\*)

alert(result); // "готово!"

}

f();

В данном примере выполнение функции остановится на строке (\*) до тех пор, пока промис не выполнится. Это произойдёт через секунду после запуска функции. После чего в переменную result будет записан результат выполнения промиса, и браузер отобразит alert-окно «готово!».

Обратите внимание, хотя await и заставляет JavaScript дожидаться выполнения промиса, это не отнимает ресурсов процессора. Пока промис не выполнится, JS-движок может заниматься другими задачами: выполнять прочие скрипты, обрабатывать события и т.п. По сути, это просто «синтаксический сахар» для получения результата промиса, более наглядный, чем promise.then.

Если использовать await внутри функции, объявленной без async, получим синтаксическую ошибку:

function f() {

let promise = Promise.resolve(1);

let result = await promise; // SyntaxError

}

Ошибки не будет, если указать ключевое слово async перед объявлением функции. Как было сказано раньше, await можно использовать только внутри async–функций.

await нельзя использовать на верхнем уровне вложенности (вне тела функции):

// SyntaxError на верхнем уровне вложенности

let response = await fetch('/article/promise-chaining/user.json');

let user = await response.json();

Можно обернуть этот код в анонимную async-функцию, тогда всё заработает:

(async () => {

let response = await fetch('/article/promise-chaining/user.json');

let user = await response.json();

...

})();

Как и promise.then, await позволяет работать с промис-совместимыми объектами. Идея в том, что если у объекта можно вызвать метод then, этого достаточно, чтобы использовать его с await. В примере ниже, экземпляры класса Thenable будут работать вместе с await:

class Thenable {

constructor(num) {

this.num = num;

}

then(resolve, reject) {

alert(resolve);

// выполнить resolve со значением this.num \* 2 через 1000мс

setTimeout(() => resolve(this.num \* 2), 1000); // (\*)

}

};

async function f() {

// код будет ждать 1 секунду,

// после чего значение result станет равным 2

let result = await new Thenable(1);

alert(result);

}

f();

Когда await получает объект с .then, не являющийся промисом, JavaScript автоматически запускает этот метод, передавая ему аргументы – встроенные функции resolve и reject. Затем await приостановит дальнейшее выполнение кода, пока любая из этих функций не будет вызвана (в примере, это строка (\*)). После чего выполнение кода продолжится с результатом resolve или reject соответственно.

Для объявления асинхронного метода достаточно написать async перед именем:

class Waiter {

async wait() {

return await Promise.resolve(1);

}

}

new Waiter()

.wait()

.then(alert); // 1

Как и с асинхронными функциями, такой метод гарантированно возвращает промис и в его теле можно использовать await.

Когда промис завершается успешно, await promise возвращает результат. Когда завершается с ошибкой – будет выброшено исключение. Как если бы на этом месте находилось выражение throw. Такой код:

async function f() {

await Promise.reject(new Error("Упс!"));

}

Делает тоже самое, что и такой:

async function f() {

throw new Error("Упс!");

}

Но есть отличие, на практике промис может завершиться с ошибкой не сразу, а через некоторое время. В этом случае будет задержка, а затем await выбросит исключение. Такие ошибки можно ловить, используя try..catch, так же как с обычным throw:

async function f() {

try {

let response = await fetch('http://no-such-url');

} catch(err) {

alert(err); // TypeError: failed to fetch

}

}

f();

В случае ошибки выполнение try прерывается и управление переходит в начало блока catch. Блоком try можно обернуть несколько строк:

async function f() {

try {

let response = await fetch('/no-user-here');

let user = await response.json();

} catch(err) {

// перехватит любую ошибку в блоке try: и в fetch, и в response.json

alert(err);

}

}

f();

Если нет try..catch, асинхронная функция будет возвращать завершившийся с ошибкой промис (в состоянии rejected). В этом случае можно использовать метод .catch промиса, чтобы обработать ошибку:

async function f() {

let response = await fetch('http://no-such-url');

}

// f() вернёт промис в состоянии rejected

f().catch(alert); // TypeError: failed to fetch // (\*)

Если не добавить .catch, то будет сгенерирована ошибка «Uncaught promise error» и информация об этом будет выведена в консоль.

При работе с async/await, .then используется нечасто, так как await автоматически ожидает завершения выполнения промиса. В этом случае, обычно (но не всегда) гораздо удобнее перехватывать ошибки используя try..catch, нежели чем .catch. Но на верхнем уровне вложенности (вне async–функций) await использовать нельзя, поэтому .then/catchдля обработки финального результата или ошибок – обычная практика. Так сделано в строке (\*) в примере выше.

Когда необходимо подождать несколько промисов одновременно, можно обернуть их в Promise.all, и затем await:

// await будет ждать массив с результатами выполнения всех промисов

let results = await Promise.all([

fetch(url1),

fetch(url2),

...

]);

В случае ошибки она будет передаваться как обычно: от завершившегося с ошибкой промиса к Promise.all. А после будет сгенерировано исключение, которое можно отловить, обернув выражение в try..catch.

Метод allSettled ведет себя иначе. Он выполняется всякий раз, когда все промисы заканчиваются, то есть выполнились успешно или выполнились с ошибкой. Он возвращает массив, который содержит как статус промиса, так и значение (или ошибку). Таким образом, allSettled никогда не отклоняется. Он либо в состоянии ожидания, либо выполнился успешно.

const promiseArray = [

Promise.resolve(200),

Promise.reject(""),

Promise.reject(new Error("Error")),

];

Promise.allSettled(promiseArray).then((result) => {

console.log("All Promises Settled", result);

});

// All Promises Settled

// [

// {status: "fulfilled", value: 200},

// {status: "rejected", reason: ""},

// {status: "rejected", reason: Error: Error at <anonymous>:4:18}

// ]

1. **Управление памятью в JavaScript.**

Управление памятью в JavaScript выполняется автоматически и незаметно. Создаются примитивы, объекты, функции. Всё это занимает память.

Основной концепцией управления памятью в JavaScript является принцип достижимости. «Достижимые» значения – это те, которые доступны или используются. Они гарантированно находятся в памяти.

1. Существует базовое множество достижимых значений, которые не могут быть удалены. Например:

* Локальные переменные и параметры текущей функции.
* Переменные и параметры других функций в текущей цепочке вложенных вызовов.
* Глобальные переменные.
* Некоторые другие внутренние значения.

Эти значения будем называть корнями.

1. Любое другое значение считается достижимым, если оно доступно из корня по ссылке или по цепочке ссылок.

Например, если в локальной переменной есть объект, и он имеет свойство, в котором хранится ссылка на другой объект, то этот объект считается достижимым. И те, на которые он ссылается, тоже достижимы. Детальные примеры на эту тему скоро последуют.

В интерпретаторе JavaScript есть фоновый процесс, который называется [сборщик мусора](https://ru.wikipedia.org/wiki/%D0%A1%D0%B1%D0%BE%D1%80%D0%BA%D0%B0_%D0%BC%D1%83%D1%81%D0%BE%D1%80%D0%B0). Он следит за всеми объектами и удаляет те, которые стали недостижимы. Простой пример:

// в user находится ссылка на объект

let user = {

name: "John"

};

Здесь стрелочка обозначает ссылку на объект. Глобальная переменная user ссылается на объект {name: "John"}. В свойстве "name" объекта John хранится примитив, поэтому оно нарисовано внутри объекта.

Если перезаписать значение user, то ссылка потеряется:

user = null;

Теперь объект John становится недостижимым. К нему нет доступа, на него нет ссылок. Сборщик мусора удалит эти данные и освободит память.

Представим, что ссылка из user  была скопирована в admin:

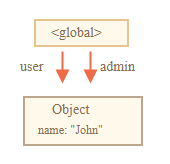
// в user находится ссылка на объект

let user = {

name: "John"

};

let admin = user;



Теперь, если перезаписать значение user:

user = null;

то объект John всё ещё достижим через глобальную переменную admin, поэтому он находится в памяти. Если также перезаписать admin, то John будет удалён.

Рассмотрим [взаимосвязанные](https://learn.javascript.ru/garbage-collection#vzaimosvyazannye-obekty) объекты:

function marry(man, woman) {

woman.husband = man;

man.wife = woman;

return {

father: man,

mother: woman

}

}

let family = marry({

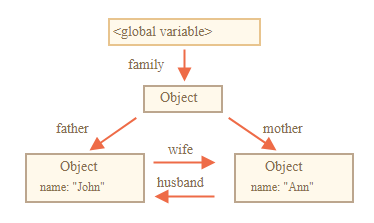
name: "John"

}, {

name: "Ann"

});

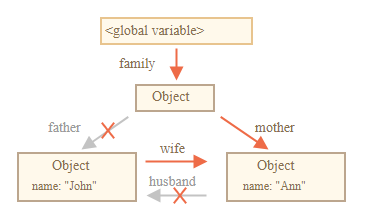
В функции marry два объекта получают ссылки друг на друга, и возвращают новый объект, содержащий ссылки на два предыдущих. В результате получается такая структура памяти:



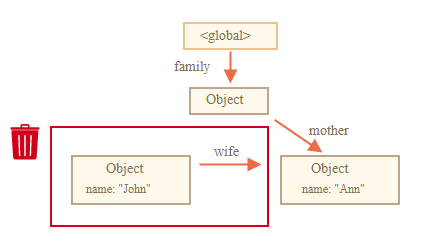
На данный момент все объекты достижимы. Удалим две ссылки:

delete family.father;

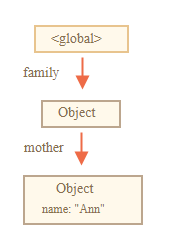
delete family.mother.husband;



Недостаточно удалить только одну из этих ссылок, потому что все объекты останутся достижимыми. Но если удалить обе, то будет видно, что у объекта John больше нет входящих ссылок:



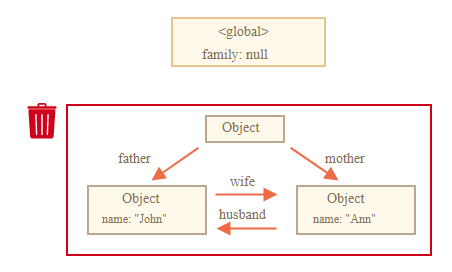
Исходящие ссылки не имеют значения. Только входящие ссылки могут сделать объект достижимым. Объект John теперь недостижим и будет удалён из памяти со всеми своими данными, которые также стали недоступны. После сборки мусора:



Вполне возможна ситуация, при которой несколько связанных объектов могут стать недостижимым и удалится из памяти. Перезапишем объект family из примера выше:

family = null;

Структура в памяти теперь станет такой:

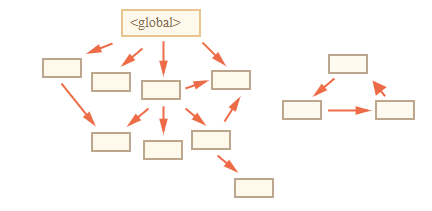


Этот пример демонстрирует, насколько важна концепция достижимости. Объекты John и Ann всё ещё связаны, оба имеют входящие ссылки, но этого недостаточно. У объекта family больше нет ссылки от корня, поэтому все эти объекты становятся недостижимым и будут удалены.

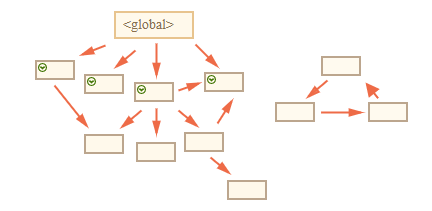
Основной алгоритм сборки мусора – «алгоритм пометок» (англ. «mark-and-sweep»). Согласно этому алгоритму, сборщик мусора регулярно выполняет следующие шаги:

* Сборщик мусора «помечает» (запоминает) все корневые объекты.
* Затем он идёт по их ссылкам и помечает все найденные объекты.
* Затем он идёт по ссылкам помеченных объектов и помечает объекты, на которые есть ссылка от них. Все объекты запоминаются, чтобы в будущем не посещать один и тот же объект дважды.
* И так далее, пока не будут посещены все ссылки (достижимые от корней).
* Все непомеченные объекты удаляются.

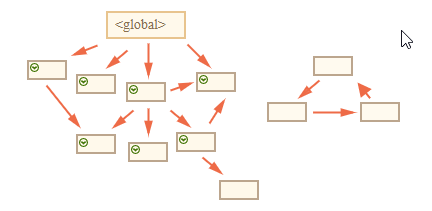
Например, пусть структура объектов выглядит так:



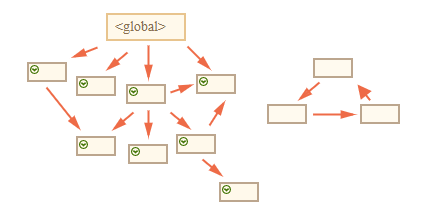
Явно видны недостижимые объекты справа. Рассмотрим, как будет работать «алгоритм пометок» сборщика мусора. На первом шаге помечаются корни:



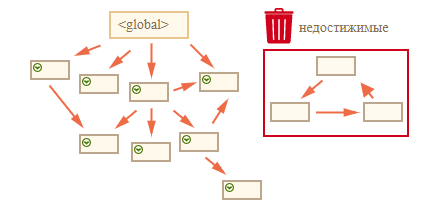
Затем помечаются объекты по их ссылкам:



Затем объекты по их ссылкам и так далее, пока это вообще возможно:



Теперь объекты, до которых не удалось дойти от корней, считаются недостижимыми и будут удалены:



Это и есть принцип работы сборки мусора.

Интерпретаторы JavaScript применяют множество оптимизаций, чтобы сборка мусора работала быстрее и не влияла на производительность.

Вот некоторые из оптимизаций:

* Сборка по поколениям (Generational collection) – объекты делятся на «новые» и «старые». Многие объекты появляются, выполняют свою задачу и быстро умирают, их можно удалять более агрессивно. Те, которые живут достаточно долго, становятся «старыми» и проверяются реже.
* Инкрементальная сборка (Incremental collection) – если объектов много, то обход всех ссылок и пометка достижимых объектов может занять значительное время и привести к видимым задержкам выполнения скрипта. Поэтому интерпретатор пытается организовать сборку мусора поэтапно. Этапы выполняются по отдельности один за другим. Это требует дополнительного учёта для отслеживания изменений между этапами, но зато теперь у нас есть много крошечных задержек вместо одной большой.
* Сборка в свободное время (Idle-time collection) – чтобы уменьшить возможное влияние на производительность, сборщик мусора старается работать только во время простоя процессора.

Существуют и другие способы оптимизации и разновидности алгоритмов сборки мусора.