**Frontend questions**

**JavaScript**

<https://developer.mozilla.org/ru/docs/Learn/Getting_started_with_the_web/JavaScript_basics>

https://www.w3schools.com/js/default.asp

https://itlogia.ru/article/neprostye\_voprosy\_javascriptsobesedovanie

[https://habr.com/ru/articles/486820/#26](https://habr.com/ru/articles/486820/" \l "26)

[Типы данных в JavaScript?](https://youtu.be/ycYp7CYOnO0?t=471)

[https://www.w3schools.com/js/js\_datatypes.asp](https://www.w3schools.com/js/js_datatypes.asp)

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Data_structures>

Значение в JavaScript всегда относится к данным определённого типа. Например, это может быть строка или число.

Есть восемь основных типов данных в JavaScript. В этой главе мы рассмотрим их в общем, а в следующих главах поговорим подробнее о каждом.

Переменная в JavaScript может содержать любые данные. В один момент там может быть строка, а в другой – число:

// Не будет ошибкой

let message = "hello";

message = 123456;

Языки программирования, в которых такое возможно, называются «динамически типизированными». Это значит, что типы данных есть, но переменные не привязаны ни к одному из них.

## [Число](https://learn.javascript.ru/types" \l "chislo)

let n = 123;

n = 12.345;

Числовой тип данных (number) представляет как целочисленные значения, так и числа с плавающей точкой.

Существует множество операций для чисел, например, умножение \*, деление /, сложение +, вычитание - и так далее.

Кроме обычных чисел, существуют так называемые «специальные числовые значения», которые относятся к этому типу данных: Infinity, -Infinity и NaN.

* Infinity представляет собой математическую [бесконечность](https://ru.wikipedia.org/wiki/Бесконечность" \l "В_математике) ∞. Это особое значение, которое больше любого числа.

Мы можем получить его в результате деления на ноль:

alert( 1 / 0 ); // Infinity

Или задать его явно:

alert( Infinity ); // Infinity

* NaN означает вычислительную ошибку. Это результат неправильной или неопределённой математической операции, например:

alert( "не число" / 2 ); // NaN, такое деление является ошибкой

Значение NaN «прилипчиво». Любая математическая операция с NaN возвращает NaN:

alert( NaN + 1 ); // NaN

alert( 3 \* NaN ); // NaN

alert( "не число" / 2 - 1 ); // NaN

Если где-то в математическом выражении есть NaN, то оно распространяется на весь результат (есть только одно исключение: NaN \*\* 0 равно 1).

**Математические операции – безопасны**

Математические операции в JavaScript «безопасны». Мы можем делать что угодно: делить на ноль, обращаться с нечисловыми строками как с числами и т.д.

Скрипт никогда не остановится с фатальной ошибкой (не «умрёт»). В худшем случае мы получим NaN как результат выполнения.

Специальные числовые значения относятся к типу «число». Конечно, это не числа в привычном значении этого слова.

Подробнее о работе с числами мы поговорим в главе [Числа](https://learn.javascript.ru/number).

## [BigInt](https://learn.javascript.ru/types" \l "bigint)

В JavaScript тип number не может безопасно работать с числами, большими, чем (253-1) (т. е. 9007199254740991) или меньшими, чем -(253-1) для отрицательных чисел. Технически, тип number может хранить и гораздо большие значения (вплоть до 1.7976931348623157 \* 10308), однако за пределами безопасного диапазона ±(253-1) многие из чисел не могут быть представлены с помощью этого типа данных из-за ограничений, вызванных внутренним представлением чисел в двоичной форме. Например, нечётные числа, большие, чем (253-1), невозможно хранить при помощи типа number, они с разной точностью будут автоматически округляться до чётных значений. В то же время некоторые чётные числа, большие, чем (253-1), при помощи типа number хранить технически возможно (однако не стоит этого делать во избежание дальнейших ошибок).

Для большинства случаев достаточно безопасного диапазона чисел от -(253-1) до (253-1). Но иногда нам нужен диапазон действительно гигантских целых чисел без каких-либо ограничений или пропущенных значений внутри него. Например, в криптографии или при использовании метки времени («timestamp») с микросекундами.

Тип BigInt был добавлен в JavaScript, чтобы дать возможность работать с целыми числами произвольной длины.

Чтобы создать значение типа BigInt, необходимо добавить n в конец числового литерала:

// символ "n" в конце означает, что это BigInt

const bigInt = 1234567890123456789012345678901234567890n;

Так как BigInt-числа нужны достаточно редко, мы рассмотрим их в отдельной главе [BigInt](https://learn.javascript.ru/bigint). Ознакомьтесь с ней, когда вам понадобятся настолько большие числа.

**Поддержка**

В данный момент BigInt поддерживается только в браузерах Firefox, Chrome, Edge и Safari, но не поддерживается в IE.

## [Строка](https://learn.javascript.ru/types" \l "stroka)

Строка (string) в JavaScript должна быть заключена в кавычки.

let str = "Привет";

let str2 = 'Одинарные кавычки тоже подойдут';

let phrase = `Обратные кавычки позволяют встраивать переменные ${str}`;

В JavaScript существует три типа кавычек.

1. Двойные кавычки: "Привет".
2. Одинарные кавычки: 'Привет'.
3. Обратные кавычки: `Привет`.

Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript.

Обратные же кавычки имеют расширенную функциональность. Они позволяют нам встраивать выражения в строку, заключая их в ${…}. Например:

let name = "Иван";

// Вставим переменную

alert( `Привет, ${name}!` ); // Привет, Иван!

// Вставим выражение

alert( `результат: ${1 + 2}` ); // результат: 3

Выражение внутри ${…} вычисляется, и его результат становится частью строки. Мы можем положить туда всё, что угодно: переменную name, или выражение 1 + 2, или что-то более сложное.

Обратите внимание, что это можно делать только в обратных кавычках. Другие кавычки не имеют такой функциональности встраивания!

alert( "результат: ${1 + 2}" ); // результат: ${1 + 2} (двойные кавычки ничего не делают)

Мы рассмотрим строки более подробно в главе [Строки](https://learn.javascript.ru/string).

**Нет отдельного типа данных для одного символа.**

В некоторых языках, например C и Java, для хранения одного символа, например "a" или "%", существует отдельный тип. В языках C и Java это char.

В JavaScript подобного типа нет, есть только тип string. Строка может содержать ноль символов (быть пустой), один символ или множество.

## [Булевый (логический) тип](https://learn.javascript.ru/types" \l "bulevyy-logicheskiy-tip)

Булевый тип (boolean) может принимать только два значения: true (истина) и false (ложь).

Такой тип, как правило, используется для хранения значений да/нет: true значит «да, правильно», а false значит «нет, не правильно».

Например:

let nameFieldChecked = true; // да, поле отмечено

let ageFieldChecked = false; // нет, поле не отмечено

Булевые значения также могут быть результатом сравнений:

let isGreater = 4 > 1;

alert( isGreater ); // true (результатом сравнения будет "да")

Мы рассмотрим булевые значения более подробно в главе [Логические операторы](https://learn.javascript.ru/logical-operators).

## [Значение «null»](https://learn.javascript.ru/types" \l "znachenie-null)

Специальное значение null не относится ни к одному из типов, описанных выше.

Оно формирует отдельный тип, который содержит только значение null:

let age = null;

В JavaScript null не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках.

Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно».

В приведённом выше коде указано, что значение переменной age неизвестно.

## [Значение «undefined»](https://learn.javascript.ru/types" \l "znachenie-undefined)

Специальное значение undefined также стоит особняком. Оно формирует тип из самого себя так же, как и null.

Оно означает, что «значение не было присвоено».

Если переменная объявлена, но ей не присвоено никакого значения, то её значением будет undefined:

let age;

alert(age); // выведет "undefined"

Технически мы можем присвоить значение undefined любой переменной:

let age = 123;

// изменяем значение на undefined

age = undefined;

alert(age); // "undefined"

…Но так делать не рекомендуется. Обычно null используется для присвоения переменной «пустого» или «неизвестного» значения, а undefined – для проверок, была ли переменная назначена.

## [Объекты и символы](https://learn.javascript.ru/types" \l "obekty-i-simvoly)

Тип object (объект) – особенный.

Все остальные типы называются «примитивными», потому что их значениями могут быть только простые значения (будь то строка, или число, или что-то ещё). В объектах же хранят коллекции данных или более сложные структуры.

Объекты занимают важное место в языке и требуют особого внимания. Мы разберёмся с ними в главе [Объекты](https://learn.javascript.ru/object) после того, как узнаем больше о примитивах.

Тип symbol (символ) используется для создания уникальных идентификаторов в объектах. Мы упоминаем здесь о нём для полноты картины, изучим этот тип после объектов.

## [Оператор typeof](https://learn.javascript.ru/types" \l "type-typeof)

Оператор typeof возвращает тип аргумента. Это полезно, когда мы хотим обрабатывать значения различных типов по-разному или просто хотим сделать проверку.

У него есть две синтаксические формы:

// Обычный синтаксис

typeof 5 // Выведет "number"

// Синтаксис, напоминающий вызов функции (встречается реже)

typeof(5) // Также выведет "number"

Если передается выражение, то нужно заключать его в скобки, т.к. typeof имеет более высокий приоритет, чем бинарные операторы:

typeof 50 + " Квартир"; // Выведет "number Квартир"

typeof (50 + " Квартир"); // Выведет "string"

Другими словами, скобки необходимы для определения типа значения, которое получилось в результате выполнения выражения в них.

Вызов typeof x возвращает строку с именем типа:

typeof undefined // "undefined"

typeof 0 // "number"

typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object" (1)

typeof null // "object" (2)

typeof alert // "function" (3)

Последние три строки нуждаются в пояснении:

1. Math — это встроенный объект, который предоставляет математические операции и константы. Мы рассмотрим его подробнее в главе [Числа](https://learn.javascript.ru/number). Здесь он служит лишь примером объекта.
2. Результатом вызова typeof null является "object". Это официально признанная ошибка в typeof, ведущая начало с времён создания JavaScript и сохранённая для совместимости. Конечно, null не является объектом. Это специальное значение с отдельным типом.
3. Вызов typeof alert возвращает "function", потому что alert является функцией. Мы изучим функции в следующих главах, где заодно увидим, что в JavaScript нет специального типа «функция». Функции относятся к объектному типу. Но typeof обрабатывает их особым образом, возвращая "function". Так тоже повелось от создания JavaScript. Формально это неверно, но может быть удобным на практике.

## [Итого](https://learn.javascript.ru/types" \l "itogo)

В JavaScript есть 8 основных типов данных.

* Семь из них называют «примитивными» типами данных:
  + number для любых чисел: целочисленных или чисел с плавающей точкой; целочисленные значения ограничены диапазоном ±(253-1).
  + bigint для целых чисел произвольной длины.
  + string для строк. Строка может содержать ноль или больше символов, нет отдельного символьного типа.
  + boolean для true/false.
  + null для неизвестных значений – отдельный тип, имеющий одно значение null.
  + undefined для неприсвоенных значений – отдельный тип, имеющий одно значение undefined.
  + symbol для уникальных идентификаторов.
* И один не является «примитивным» и стоит особняком:
  + object для более сложных структур данных.

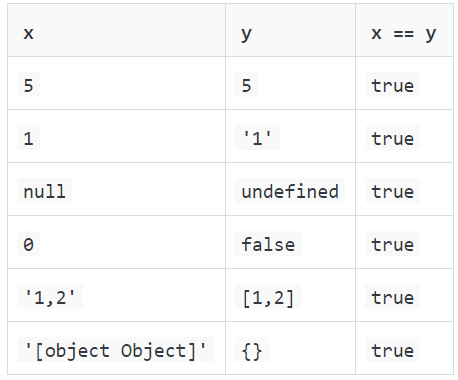
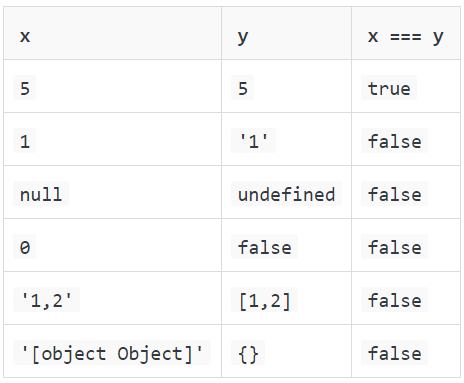
Оператор typeof позволяет нам увидеть, какой тип данных сохранён в переменной.

* Имеет две формы: typeof x или typeof(x).
* Возвращает строку с именем типа. Например, "string".
* Для null возвращается "object" – это ошибка в языке, на самом деле это не объект.

[Разница между == и === (нестрогое/строгое равенство)?](https://youtu.be/ycYp7CYOnO0?t=529)

Разница между оператором "==" (абстрактное или нестрогое равенство) и оператором "===" (строгое равенство) состоит в том, что первый сравнивает значения после их преобразования или приведения к одному типу (Coersion), а второй — без такого преобразования.  
  
Давайте копнем глубже. И сначала поговорим о преобразовании.  
  
Преобразование представляет собой процесс приведения значения к другому типу или, точнее, процесс приведения сравниваемых значений к одному типу. При сравнении оператор "==" производит так называемое неявное сравнение. Оператор "==" выполняет некоторые операции перед сравнением двух значений.  
  
Допустим, мы сравниваем x и y.  
  
Алгоритм следующий:

1. Если x и y имеют одинаковый тип, сравнение выполняется с помощью оператора "===".
2. Если x = null и y = undefined возвращается true.
3. Если x = undefined и y = null возвращается true.
4. Если x = число, а y = строка, возвращается x == toNumber(y) (значение y преобразуется в число).
5. Если x = строка, а y = число, возвращается toNumber(x) == y (значение x преобразуется в число).
6. Если x = логическое значение, возвращается toNumber(x) == y.
7. Если y = логическое значение, возвращается x == toNumber(y).
8. Если x = строка, символ или число, а y = объект, возвращается x == toPrimitive(y) (значение y преобразуется в примитив).
9. Если x = объект, а y = строка, символ или число, возвращается toPrimitive(x) == y.
10. Возвращается false.

Запомните: для приведения объекта к «примитиву» метод toPrimitive сначала использует метод valueOf, затем метод toString.  
  
Примеры:  
  
  
  
Все примеры возвращают true.  
  
Первый пример — первое условие алгоритма.  
Второй пример — четвертое условие.  
Третий — второе.  
Четвертый — седьмое.  
Пятый — восьмое.  
И последний — десятое.  
  
  
  
Если же мы используем оператор "===" все примеры, кроме первого, вернут false, поскольку значения в этих примерах имеют разные типы.

[Что такое Strict mode в JavaScript?](https://youtu.be/ycYp7CYOnO0?t=577)

**Strict mode** (в переводе с анг. - строгий режим) – это модель синтаксического анализа и выполнения JavaScript-кода, в которой используется более строгая проверка кода на ошибки и генерируются ошибки при небезопасных действиях.

Строгий режим принёс ряд изменений в обычную семантику JavaScript. Во-первых, строгий режим заменяет исключениями некоторые ошибки, которые интерпретатор JavaScript ранее молча пропускал. Во-вторых, строгий режим исправляет ошибки, которые мешали движкам JavaScript выполнять оптимизацию -- в некоторых случаях код в строгом режиме может быть оптимизирован для более быстрого выполнения, чем код в обычном режиме. В-третьих, строгий режим запрещает использовать некоторые элементы синтаксиса, которые, вероятно, в следующих версиях ECMAScript получат особый смысл.

## Включение строгого режима

Чтобы включить строгий режим для всего сценария, добавьте в начало сценария следующую команду:

|  |  |
| --- | --- |
| 1 | "use strict"; |

Хоть она и выглядит как строка, на самом деле это директива, переводящая JavaScript в строгий режим.

Строгий режим можно включить и для отдельной функции, добавив эту директиву в начало тела функции:

|  |  |
| --- | --- |
| 1  2  3 | **function** foo() {    "use strict";  } |

Директиву "use strict" можно указывать только в самом начале сценария или в самом начале функции. Это не означает, что она должна находиться в самой первой строке, но должна быть указана до других инструкций. Если директиву "use strict" указать не в начале, а в другом месте, она не будет работать:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | "use strict";   // Правильное расположение  a = 5;  alert(a);  b = 1;  alert(b);    a = 5;  alert(a);  "use strict";   // Неправильное расположение. Директива не будет работать  b = 1;  alert(b); |

## Выполнение JavaScript в строгом режиме

* Все [переменные](https://puzzleweb.ru/javascript/4_variables.php) перед их использованием должны быть объявлены. Попытка использовать переменную до [объявления](_blank) вызывает исключение.
* При попытке присвоить значение необъявленной переменной возникает ошибка. Однако явное создание [свойства глобального объекта](https://puzzleweb.ru/javascript/obj_global.php) и присвоение ему значения ошибку не вызывает:

|  |  |
| --- | --- |
| 1 | window.num = 10 |

* В качестве идентификаторов нельзя использовать следующие слова: eval и arguments.
* Попытка добавить новое свойство примитивному значению вызывает исключение.
* Нельзя использовать инструкцию [with](https://puzzleweb.ru/javascript/14_objects2.php).
* [Параметры функции](https://puzzleweb.ru/javascript/13_functions2.php) не могут иметь одинаковые имена.
* Параметры функции не синхронизируются со значениями объекта [arguments](_blank).
* Если функция вызывается как функция (а не как метод), ключевое слово [this](_blank) в качестве значения получает значение [undefined](_blank), а не ссылку на глобальный объект.
* Код, переданный функции [eval()](https://puzzleweb.ru/javascript/global_eval.php), выполняется в песочнице, это означает, что код по-прежнему может читать и изменять уже существующие переменные, однако переменные и функции, объявленные в переданном коде, наружу из песочницы не попадут:

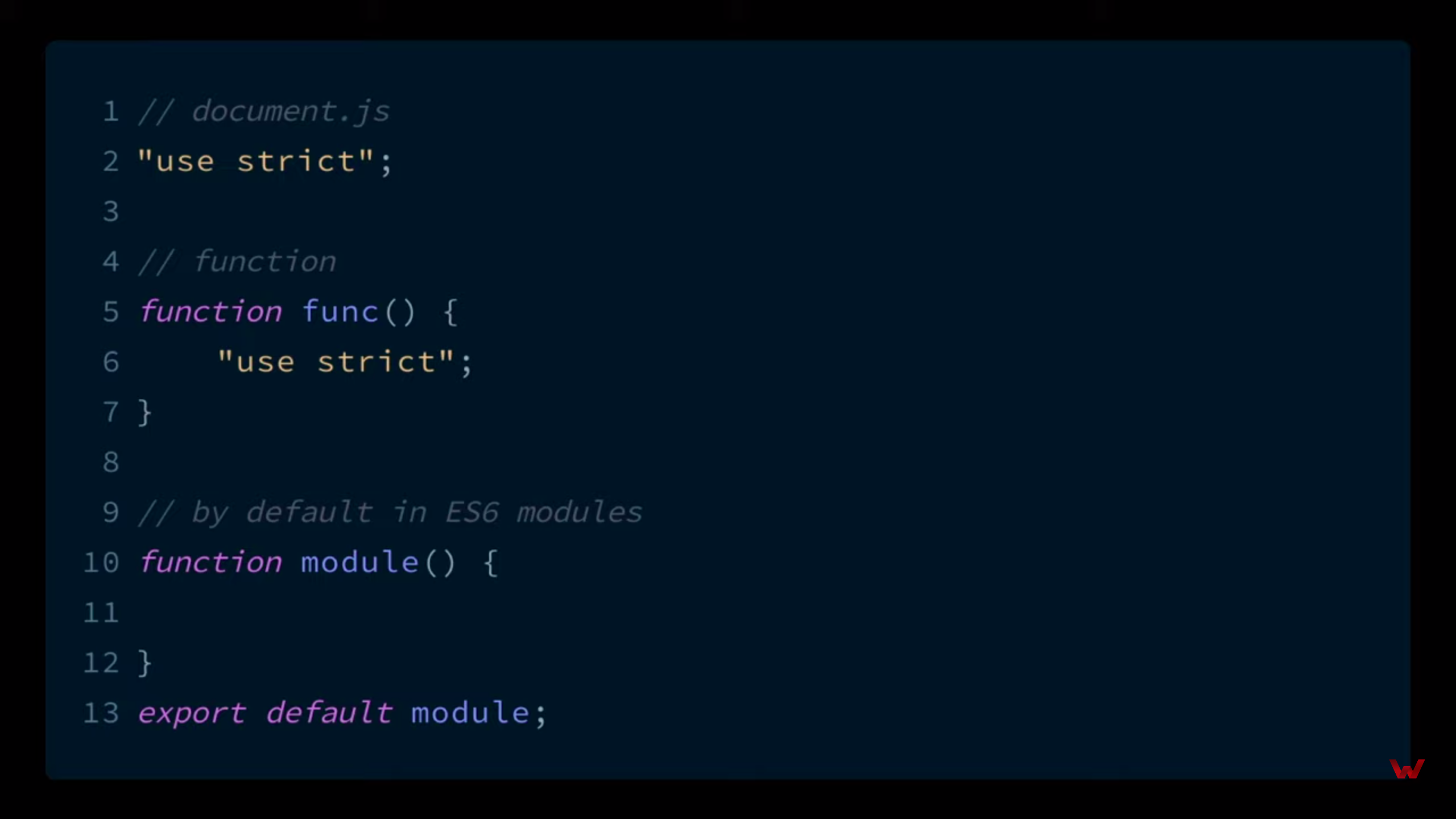
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | "use strict";  **function** foo() {  **var** a = 10;    eval("var x = 10; alert(x + a);");      alert(x);   // Ошибка. Такой переменной не существует  }    foo(); |

* Операндом оператора [delete](https://puzzleweb.ru/javascript/7_delete.php) может быть только настраиваемое свойство объекта или элемент массива. Если операндом оператора delete является неквалифицированный идентификатор, такой как имя переменной, функции, параметр функции или имя ненастраиваемого свойства возбуждается ошибка.
* В цепочке присваиваний нельзя использовать необъявленные переменные:

|  |  |
| --- | --- |
| 1 | **var** a = b = 5;   // Ошибка. Необъявленная переменная (b) |

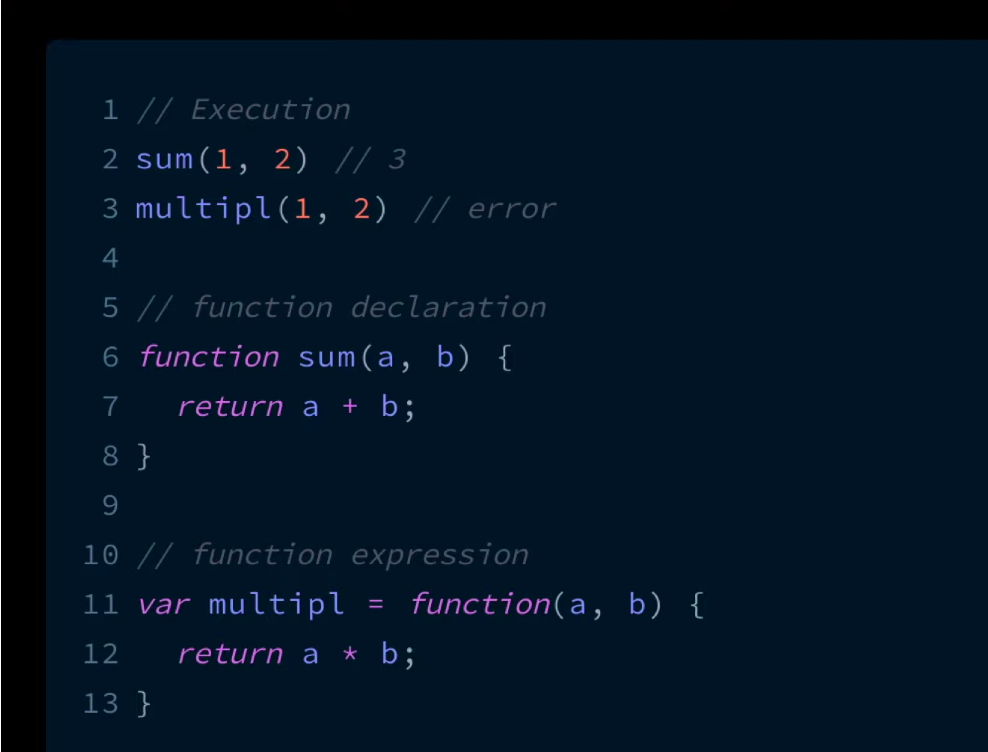
* Попытка присвоить значение свойству, недоступному для записи, или создание нового свойства в нерасширяемом объекте порождают исключение.
* Обращение к свойствам arguments.caller и arguments.callee порождает исключение.
* Функции, объявленные как FD внутри блока, находятся в блочной области видимости:

|  |  |
| --- | --- |
| 1  2  3  4  5 | {    foo();   // 1  **function** foo () { alert(1); }  }  foo();     // Ошибка. Функция foo не объявлена. |



[Разница между function declaration и function expression?](https://youtu.be/ycYp7CYOnO0?t=632)

Function declaration создана в основном потоке документа, создается интерпретатором до выполнения кода, поэтому ее можно вызвать до обьявления. Это происходит благодаря hoisting – всплытию.



[More info](https://learn.javascript.ru/function-expressions" \l "function-expression-v-sravnenii-s-function-declaration)

Давайте разберём ключевые отличия Function Declaration от Function Expression.

Во-первых, синтаксис: как отличить их друг от друга в коде.

* Function Declaration: функция объявляется отдельной конструкцией «function…» в основном потоке кода.

// Function Declaration

function sum(a, b) {

return a + b;

}

* Function Expression: функция, созданная внутри другого выражения или синтаксической конструкции. В данном случае функция создаётся в правой части «выражения присваивания» =:

// Function Expression

let sum = function(a, b) {

return a + b;

};

Более тонкое отличие состоит в том, когда создаётся функция движком JavaScript.

Function Expression создаётся, когда выполнение доходит до него, и затем уже может использоваться.

После того, как поток выполнения достигнет правой части выражения присваивания let sum = function… – с этого момента, функция считается созданной и может быть использована (присвоена переменной, вызвана и т.д. ).

С Function Declaration всё иначе.

Function Declaration может быть вызвана раньше, чем она объявлена.

Другими словами, когда движок JavaScript готовится выполнять скрипт или блок кода, прежде всего он ищет в нём Function Declaration и создаёт все такие функции. Можно считать этот процесс «стадией инициализации».

И только после того, как все объявления Function Declaration будут обработаны, продолжится выполнение.

В результате функции, созданные как Function Declaration, могут быть вызваны раньше своих определений.

Функции, объявленные при помощи Function Expression, создаются тогда, когда выполнение доходит до них. Это случится только на строке, помеченной звёздочкой (\*). Слишком поздно.

Ещё одна важная особенность Function Declaration заключается в их блочной области видимости.

В строгом режиме, когда Function Declaration находится в блоке {...}, функция доступна везде внутри блока. Но не снаружи него.

Для примера давайте представим, что нам нужно объявить функцию welcome() в зависимости от значения переменной age, которое мы получим во время выполнения кода. И затем запланируем использовать её когда-нибудь в будущем.

[Разница между null и undefined?](https://youtu.be/G7hLwudGWL4?t=511)

Для начала давайте поговорим о том, что у них общего.  
  
Во-первых, они принадлежат к 7 «примитивам» (примитивным типам) JS:

**let** primitiveTypes = ['string', 'number', 'null', 'undefined', 'boolean', 'symbol', 'bigint']

Во-вторых, они являются ложными значениями, т.е. результатом их преобразования в логическое значение с помощью Boolean() или оператора "!!" является false:

console.log(!!null) // false

console.log(!!undefined) // false

console.log(Boolean(null)) // false

console.log(Boolean(undefined)) // false

Ладно, теперь о различиях.  
  
undefined («неопределенный») представляет собой значение по умолчанию:

* переменной, которой не было присвоено значения, т.е. объявленной, но не инициализированной переменной;
* функции, которая ничего не возвращает явно, например, console.log(1);
* несуществующего свойства объекта.

В указанных случаях движок JS присваивает значение undefined.

**let** \_thisIsUndefined

**const** doNothing = () => {}

**const** someObj = {

a: 'ay',

b: 'bee',

c: 'si'

}

console.log(\_thisIsUndefined) // undefined

console.log(doNothing()) // undefined

console.log(someObj['d']) // undefined

null — это «значение отсутствия значения». null — это значение, которое присваивается переменной явно. В примере ниже мы получаем null, когда метод fs.readFile отрабатывает без ошибок:

fs.readFile('path/to/file', (e, data) => {

console.log(e) // здесь мы получаем null

**if**(e) {

console.log(e)

}

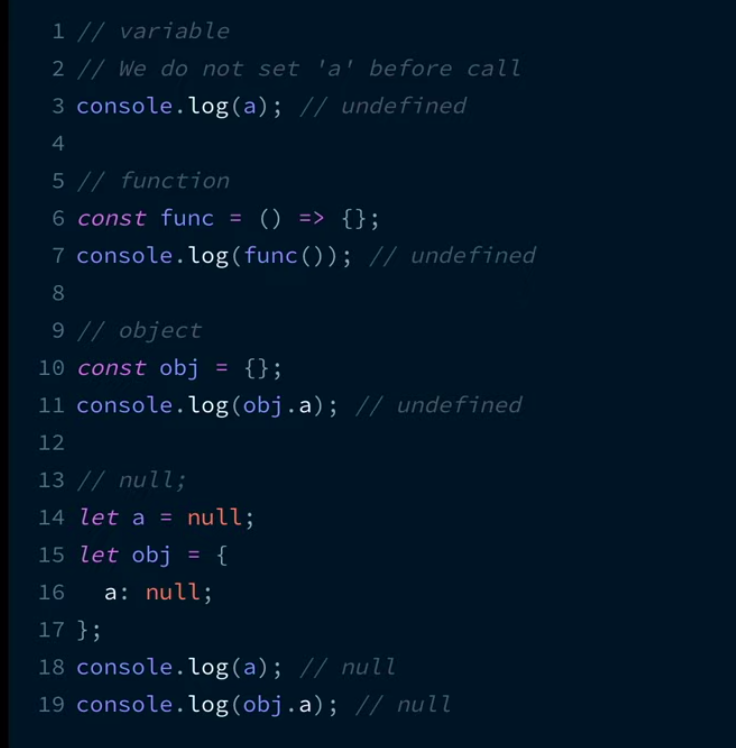
console.log(data)

})

При сравнении null и undefined мы получаем true, когда используем оператор "==", и false при использовании оператора "===". О том, почему так происходит, см. ниже.

console.log(null == undefined) // true

console.log(null === undefined) // false



[Типы таймеров в JavaScript?](https://youtu.be/G7hLwudGWL4?t=690)



**Функция setTimeout**

Для одноразового выполнения действий через промежуток времени предназначена функция setTimeout(). Она может принимать два параметра:

var timerId = setTimeout(someFunction, period)

Параметр period указывает на промежуток, через который будет выполняться функция из параметра someFunction. А в качестве результата функция возвращает id таймера.

function timerFunction() {

document.write("выполнение функции setTimeout");

}

setTimeout(timerFunction, 3000);

В данном случае через 3 секунды после загрузки страницы произойдет срабатывание функции timerFunction.

Для остановки таймера применяется функция clearTimeout().

function timerFunction() {

document.write("выполнение функции setTimeout");

}

var timerId = setTimeout(timerFunction, 3000);

clearTimeout(timerId);

**Функция setInterval**

Функции setInterval() и clearInterval() работают аналогично функциям setTimeout() и clearTimeout() с той лишь разницей, что setInterval() постоянно выполняет определенную функцию через промежуток времени.

Например, напишем небольшую программу для вывода текущего времени:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

</head>

<body>

<div id="time"></div>

<script>

function updateTime() {

document.getElementById("time").innerHTML = new Date().toTimeString();

}

setInterval(updateTime, 1000);

</script>

</body>

</html>

Здесь через каждую секунду (1000 миллисекунд) вызывается функция updateTime(), которая обновляет содержимое поля <div id="time" >, устанавливая в качестве его кода html текущее вемя.

[Что такое поднятие (Hoisting)?](https://youtu.be/G7hLwudGWL4?t=552)

Поднятие — это термин, описывающий подъем переменной или функции в глобальную или функциональную области видимости.  
  
Для того, чтобы понять, что такое Hoisting, необходимо разобраться с тем, что представляет собой контекст выполнения.  
  
Контекст выполнения — это среда, в которой выполняется код. Контекст выполнения имеет две фазы — компиляция и собственно выполнение.  
  
Компиляция. В этой фазе функциональные выражения и переменные, объявленные с помощью ключевого слова «var», со значением undefined поднимаются в самый верх глобальной (или функциональной) области видимости (как бы перемещаются в начало нашего кода. Это объясняет, почему мы можем вызывать функции до их объявления — прим. пер.).  
  
Выполнение. В этой фазе переменным присваиваются значения, а функции (или методы объектов) вызываются или выполняются.  
  
Запомните: поднимаются только функциональные выражения и переменные, объявленные с помощью ключевого слова «var». Обычные функции и стрелочные функции, а также переменные, объявленные с помощью ключевых слов «let» и «const» не поднимаются.  
  
Предположим, что у нас есть такой код:

console.log(y)

y = 1

console.log(y)

console.log(greet('Mark'))

**function** **greet**(name) {

**return** 'Hello ' + name + '!'

}

**var** y

Получаем undefined, 1 и 'Hello Mark!'.  
  
Вот как выглядит фаза компиляции:

**function** **greet**(name) {

**return** 'Hello ' + name + '!'

}

**var** y // присваивается undefined

// ожидается завершение фазы компиляции

// затем начинается фаза выполнения

/\*

console.log(y)

y = 1

console.log(y)

console.log(greet('Mark'))

\*/

После завершения фазы компиляции начинается фаза выполнения, когда переменным присваиваются значения и вызываются функции.  
  
Дополнительно о Hoisting можно почитать [здесь](https://developer.mozilla.org/ru/docs/Словарь/Поднятие).

[Статья о поднятии](https://medium.com/@stasonmars/разбираемся-с-поднятием-hoisting-в-javascript-7d2d27bc51f1" \l ":~:text=Поднятие или hoisting — это механизм,тем%2C как код будет выполнен.)

## The let and const Keywords

Variables defined with let and const are hoisted to the top of the block, but not initialized.

Meaning: The block of code is aware of the variable, but it cannot be used until it has been declared.

Using a let variable before it is declared will result in a ReferenceError.

The variable is in a "temporal dead zone" from the start of the block until it is declared:

Using a const variable before it is declared, is a syntax error, so the code will simply not run.

Read more about let and const in [JS Let / Const](https://www.w3schools.com/js/js_let.asp).

[**Что такое область видимости (Scope)?**](https://youtu.be/1eIRTdgzHtw?t=282)

Область видимости — это место, где (или откуда) мы имеем доступ к переменным или функциям. JS имеем три типа областей видимости: глобальная, функциональная и блочная (ES6).  
  
Глобальная область видимости — переменные и функции, объявленные в глобальном пространстве имен, имеют глобальную область видимости и доступны из любого места в коде.

// глобальное пространство имен

**var** g = 'global'

**function** **globalFunc**() {

**function** **innerFunc**() {

console.log(g) // имеет доступ к переменной g, поскольку она является глобальной

}

innerFunc()

}

Функциональная область видимости (область видимости функции) — переменные, функции и параметры, объявленные внутри функции, доступны только внутри этой функции.

**function** **myFavouriteFunc**(a) {

**if** (true) {

**var** b = 'Hello ' + a

}

**return** b

}

myFavouriteFunc('World')

console.log(a) // Uncaught ReferenceError: a is not defined

console.log(b) // не выполнится

Блочная область видимости — переменные (объявленные с помощью ключевых слов «let» и «const») внутри блока ({ }), доступны только внутри него.

**function** **testBlock**() {

**if** (true) {

**let** z = 5

}

**return** z

}

testBlock() // Uncaught ReferenceError: z is not defined

Область видимости — это также набор правил, по которым осуществляется поиск переменной. Если переменной не существует в текущей области видимости, ее поиск производится выше, во внешней по отношению к текущей области видимости. Если и во внешней области видимости переменная отсутствует, ее поиск продолжается вплоть до глобальной области видимости. Если в глобальной области видимости переменная обнаружена, поиск прекращается, если нет — выбрасывается исключение. Поиск осуществляется по ближайшим к текущей областям видимости и останавливается с нахождением переменной. Это называется цепочкой областей видимости (Scope Chain).

// цепочка областей видимости

// внутренняя область видимости -> внешняя область видимости -> глобальная область видимости

// глобальная область видимости

**var** variable1 = 'Comrades'

**var** variable2 = 'Sayonara'

**function** **outer**() {

// внешняя область видимости

**var** variable1 = 'World'

**function** **inner**() {

// внутренняя область видимости

**var** variable2 = 'Hello'

console.log(variable2 + ' ' + variable1)

}

inner()

}

outer()

// в консоль выводится 'Hello World',

// потому что variable2 = 'Hello' и variable1 = 'World' являются ближайшими

// к внутренней области видимости переменными

https://www.w3schools.com/js/js\_scope.asp



[More info incl about modules](https://habr.com/ru/articles/517338/)

важно что, прежде всего, области видимости вложены друг в друга, и текущий выполняемый код имеет доступ ко всем наружным областям видимости из текущей точки, корневая область из которых является глобальной областью видимости. Области видимости – это также набор правил, по которым происходит поиск переменных: сначала переменная ищется в локальной области видимости, если ее нет, то поиск происходит во внешней области, и т.д., пока поиск не доходит до глобальной области.

[**Разница между var, let и const?**](https://youtu.be/1eIRTdgzHtw?t=362)

Переменные, объявленные с помощью ключевого слова «var», являются глобальными. Это означает, что они доступны из любого места в коде:

**function** **giveMeX**(showX){

**if**(showX){

**var** x = 5

}

**return** x

}

console.log(giveMeX(false))

console.log(giveMeX(true))

Результатом первого console.log будет undefined, второго — 5. Мы имеем доступ к переменной «x» из-за ее всплытия в глобальную область видимости. Код из примера выше интерпретируется следующим образом:

**function** **giveMeX**(showX){

**var** x // имеет значение undefined

**if**(showX){

x = 5

}

**return** x

}

Результатом первого console.log является undefined, поскольку объявленные переменные, которым не присвоено значения, имеют значение undefined по умолчанию.  
  
Переменные, объявленные с помощью ключевых слов «let» и «const» имеют блочную область видимости. Это означает, что они доступны только внутри блока ({ }):

**function** **giveMeX**(showX){

**if**(showX){

**let** x = 5

}

**return** x

}

**function** **giveMeY**(showY){

**if**(showY){

**let** y = 5

}

**return** y

}

Вызов этих функций с параметром false приведет к ошибке ReferenceError, потому что к переменным «x» и «y» нет доступа снаружи блока и их значения не возвращаются (не всплывают).  
  
Разница между «let» и «const» состоит в том, что в первом случае мы может менять значение переменной, а во втором — нет (константа). При этом, мы можем менять значение свойства объекта, объявленного с помощью const, но не само свойство (переменную).



[**Что такое замыкание (Closure)?**](https://youtu.be/kx3dR6ztICU?t=284)  
По сути, замыкание — это способность функции во время создания запоминать ссылки на переменные и параметры, находящиеся в текущей области видимости, в области видимости родительской функции, в области видимости родителя родительской функции и так до глобальной области видимости с помощью цепочки областей видимости. Обычно область видимости определяется при создании функции.  
  
Примеры — отличный способ объяснить замыкание:

// глобальная область видимости

**var** globalVar = 'abc'

**function** **a**() {

// область видимости функции

console.log(globalVar)

}

a() // 'abc'

// цепочка областей видимости

// область видимости функции a -> глобальная область видимости

В данном примере, когда мы объявляем функцию, глобальная область видимости является частью замыкания.  
  
  
  
Переменная «globalVar» не имеет значения на картинке, потому что ее значение может меняться в зависимости от того, где и когда будет вызвана функция. Но в примере выше globalVar будет иметь значение «abc».  
  
Теперь пример посложнее:

**var** globalVar = 'global'

**var** outerVar = 'outer'

**function** **outerFunc**(outerParam) {

**function** **innerFunc**(innerParam) {

console.log(globalVar, outerParam, innerParam)

}

**return** innerFunc

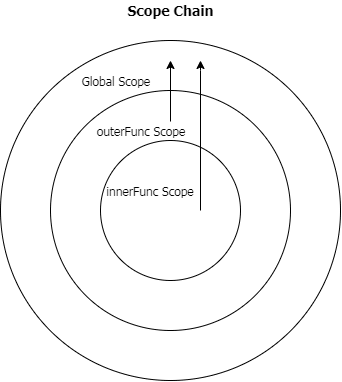
}

**const** x = outerFunc(outerVar)

outerVar = 'outer-2'

globalVar = 'guess'

x('inner')

  
  
В результате получаем «guess outer inner». Объяснение следующее: когда мы вызываем функцию outerFunc и присваиваем переменной «x» значение, возвращаемое функцией innerFunc, параметр «outerParam» равняется «outer». Несмотря на то, что мы присвоили переменной «outerVar» значение «outer-2», это произошло после вызова функции outerFunc, которая «успела» найти значение переменной «outerVar» в цепочке областей видимости, этим значением было «outer». Когда мы вызываем «x», которая ссылается на innerFunc, значением «innerParam» является «inner», потому что мы передаем это значение в качестве параметра при вызове «x». globalVar имеет значение «guess», потому что мы присвоили ей это значение перед вызовом «x».  
  
Пример неправильного понимания замыкания.

**const** arrFunc = []

**for** (**var** i = 0; i < 5; i++) {

arrFunc.push(**function**() {

**return** i

})

}

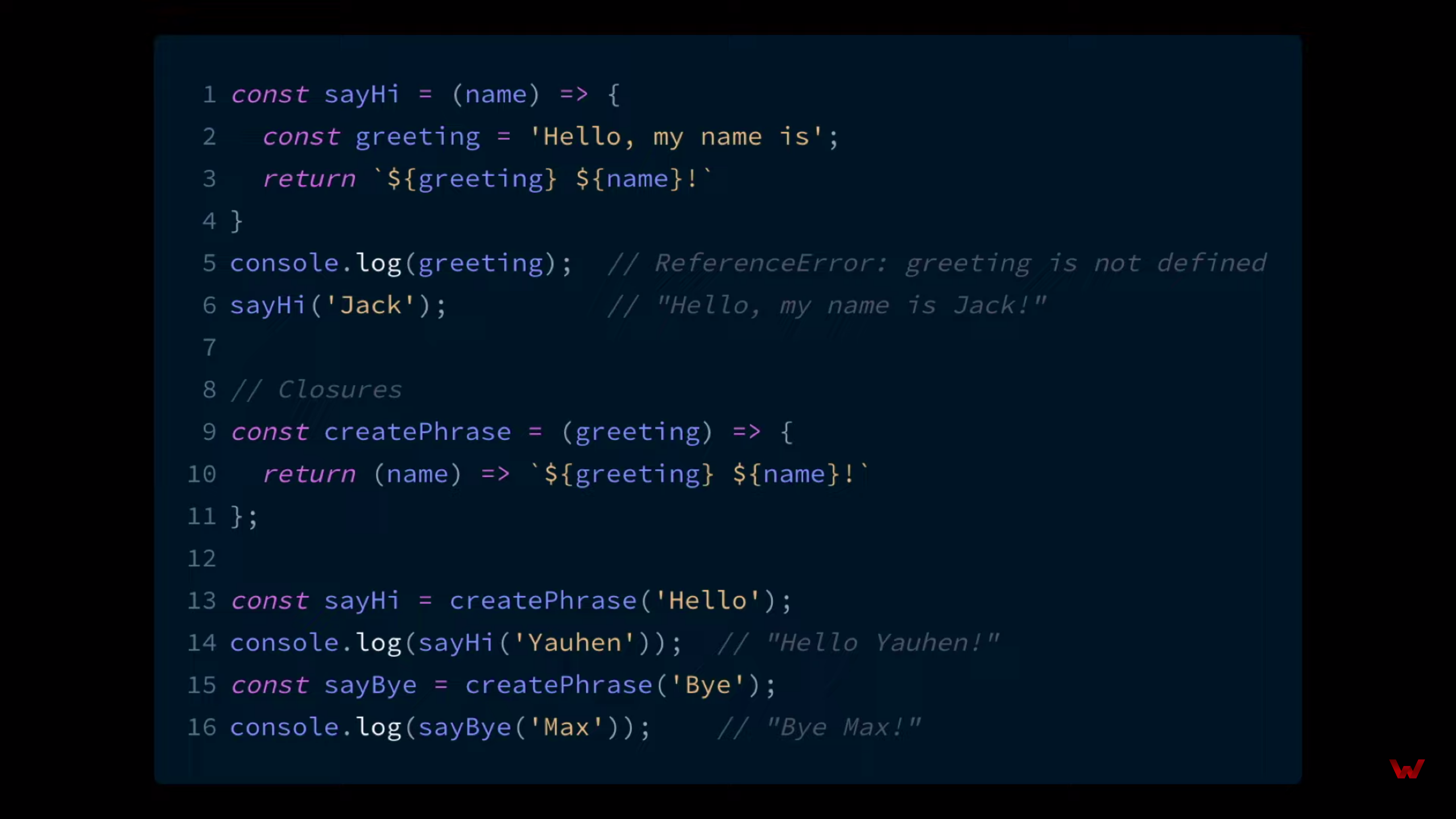
console.log(i) // 5

**for** (**let** i = 0; i < arrFunc.length; i++) {

console.log(arrFunc[i]()) // все 5

}

Данный код работает не так, как ожидается. Объявление переменной с помощью ключевого слова «var» делает эту переменную глобальной. После добавления функций в массив «arrFunc» значением глобальной переменной «i» становится «5». Поэтому когда мы вызываем функцию, она возвращает значение глобальной переменной «i». Замыкание хранит ссылку на переменную, а не на ее значение во время создания. Эту проблему можно решить, используя IIFE или объявив переменную с помощью ключевого слова «let».  
  
Подробнее о замыкании можно почитать [здесь](https://developer.mozilla.org/ru/docs/Web/JavaScript/Closures) и [здесь](https://learn.javascript.ru/closure).

При создании функции и использовании внутри нее переменных эти переменные доступны только локально внутри этой функции, снаружи мы не можем получить к ним доступ. На хранение таких переменных, в том числе и аргументов, выделяется определенная память. И когда функция заканчивает свое выполнение, эта память очищается. Таким образом, эти переменные больше нигде не существуют. Описанная логика – это логика функции sayHi.

Но если внутри одной функции создать вторую, то вложенная функция получит доступ к переменным, которые были объявлены во внешней функции. Этот механизм и называется замыканием. То есть, вложенная функция замыкает на себе переменные и аргументы внешней функции. Чтобы создать замыкание, вложенную функцию необходимо вернуть как в примере createPhrase. Таким образом, при отработке внешней функции возвращается внутренняя, которая замыкается на значении внешней и не дает памяти очиститься.

[**Что обозначает this в JavaScript?**](https://youtu.be/rlWgI7AvV18?t=507)

Обычно this ссылается на значение объекта, который в данный момент выполняет или вызывает функцию. «В данный момент» означает, что значение this меняется в зависимости от контекста выполнения, от того места, где мы используем this.

**const** carDetails = {

name: 'Ford Mustang',

yearBought: 2005,

**getName**() {

**return** this.name

}

isRegistered: true

}

console.log(carDetails.getName()) // Ford Mustang

В данном случае метод getName возвращает this.name, а this ссылается на carDetails, объект, в котором выполняется getName, который является ее «владельцем».  
  
Добавим после console.log три строчки:

**var** name = 'Ford Ranger'

**var** getCarName = carDetails.getName

console.log(getCarName()) // Ford Ranger

Второй console.log выдает Ford Ranger, и это странно. Причина такого поведения заключается в том, что «владельцем» getCarName является объект window. Переменные, объявленные с помощью ключевого слова «var» в глобальной области видимости, записываются в свойства объекта window. this в глобальной области видимости ссылается на объект window (если речь не идет о строгом режиме).

console.log(getCarName === window.getCarName) // true

console.log(getCarName === this.getCarName) // true

В этом примере this и window ссылаются на один объект.  
  
Одним из способов решения данной проблемы является использование методов call или apply:

console.log(getCarName.apply(carDetails)) // Ford Mustang

console.log(getCarName.call(carDetails)) // Ford Mustang

Call и apply принимают в качестве первого аргумента объект, который будет являться значением this внутри функции.  
  
В IIFE, функциях, которые создаются в глобальном области видимости, анонимных функциях и внутренних функциях методов объекта значением this по умолчанию является объект window.

(**function**() {

console.log(this)

})() // window

**function** **iHateThis**() {

console.log(this)

}

iHateThis() // window

**const** myFavouriteObj = {

**guessThis**() {

**function** **getName**() {

console.log(this.name)

}

getName()

},

name: 'Marko Polo',

**thisIsAnnoying**(callback) {

callback()

}

}

myFavouriteObj.guessThis() // window

myFavouriteObj.thisIsAnnoying(**function**() {

console.log(this) // window

})

Существует два способа получить «Marko Polo».  
  
Во-первых, мы можем сохранить значение this в переменной:

**const** myFavoriteObj = {

**guessThis**() {

**const** self = this // сохраняем значение this в переменной self

**function** **getName**() {

console.log(self.name)

}

getName()

},

name: 'Marko Polo',

**thisIsAnnoying**(callback) {

callback()

}

}

Во-вторых, мы можем использовать стрелочную функцию:

**const** myFavoriteObj = {

**guessThis**() {

**const** getName = () => {

// копируем значение this из внешнего окружения

console.log(this.name)

}

getName()

},

name: 'Marko Polo',

**thisIsAnnoying**(callback) {

callback()

}

}

Стрелочные функции не имеют собственного значения this. Они копируют значение this из внешнего лексического окружения.

[https://www.w3schools.com/js/js\_this.asp](https://www.w3schools.com/js/js_this.asp)

[Еще инфа](https://learn.javascript.ru/object-methods" \l ":~:text=Значение this – это объект «перед,который используется для вызова метода.&text=Здесь во время выполнения кода,(ссылка на объект user ).)

# [Больше](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/this)

this — это **ключевое слово в JavaScript, которое содержит в себе объект (контекст) выполняемого кода**. Мне кажется, что проще всего представить, что this — это уникальная переменная, которая хранит в себе контекст исполняемого кода. И наоборот — контекст — это значение ключевого слова this.

Ключевое слово this — это контекст вызова или ссылка на значение объекта, который в данный момент выполняет или вызывает функцию. В соответствии с этим this может принимать абсолютно разные значения. Это может быть глобальный объект, или объявленный, или объект события.

[**Что такое функции высшего порядка (Higher Order Functions)?**](https://youtu.be/1eIRTdgzHtw?t=426)

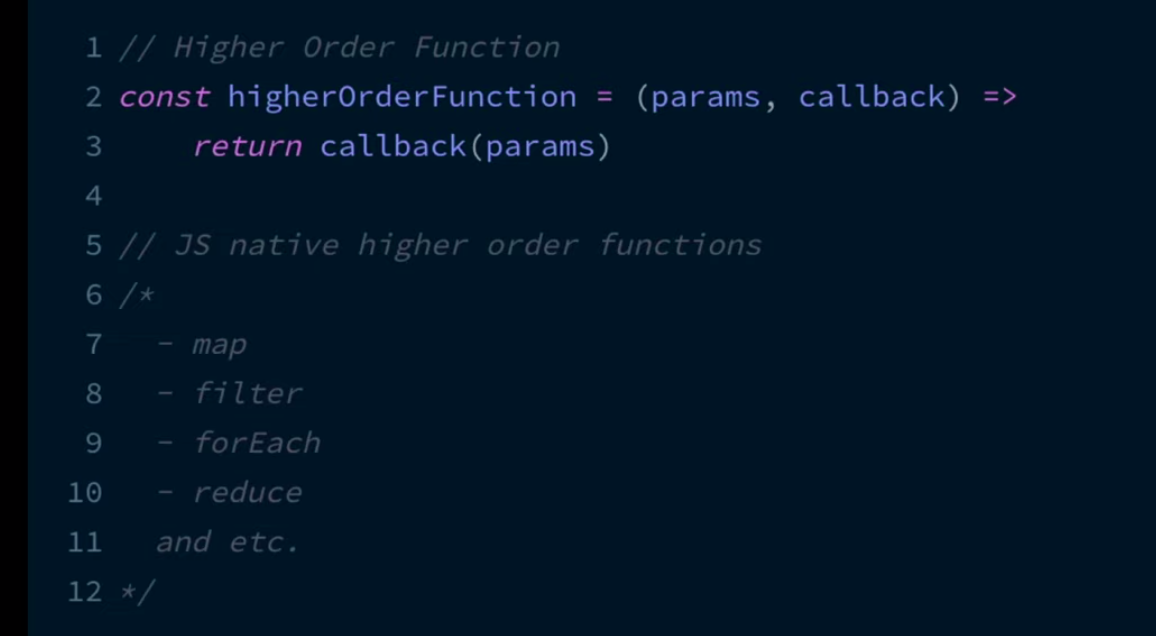
Функция высшего порядка — это функция, возвращающая другую функцию или принимающая другую функцию в качестве аргумента.

**function** **higherOrderFunction**(param, callback) {

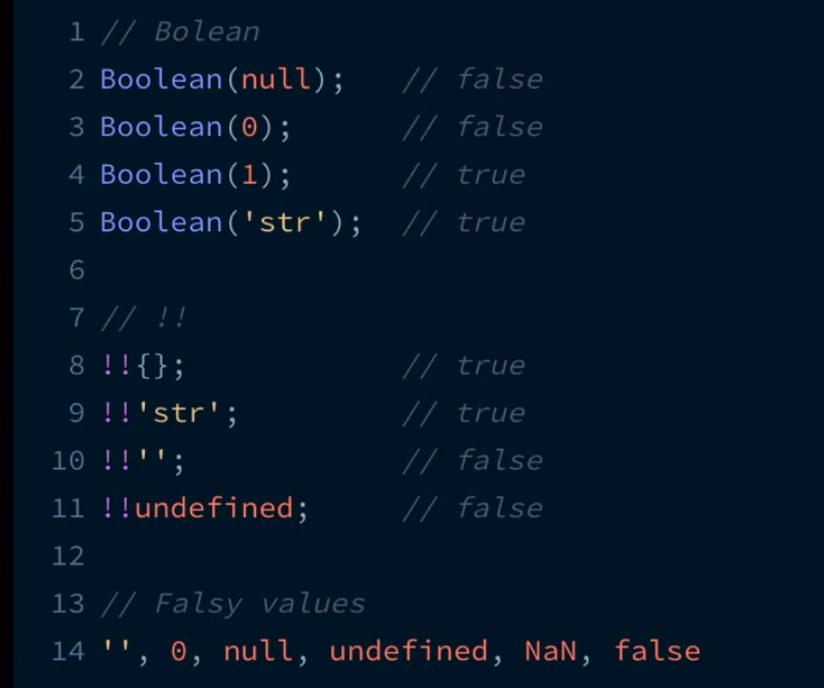
**return** callback(param)

}

https://habr.com/ru/companies/ruvds/articles/428570/



[**Как превратить любой тип данных в булевый? Перечислите ложные значения в JS?**](https://youtu.be/CjdCxxqObaM?t=368)



[**Методы строк в JavaScript?**](https://youtu.be/CjdCxxqObaM?t=415)

Slice, substring — указываются индексы, с которых начинается и на которых заканчивается подстрока (не включая), но subsctring не может принимать отрицательные значения. Subsctr в качестве второго параметра принимает количество символов.

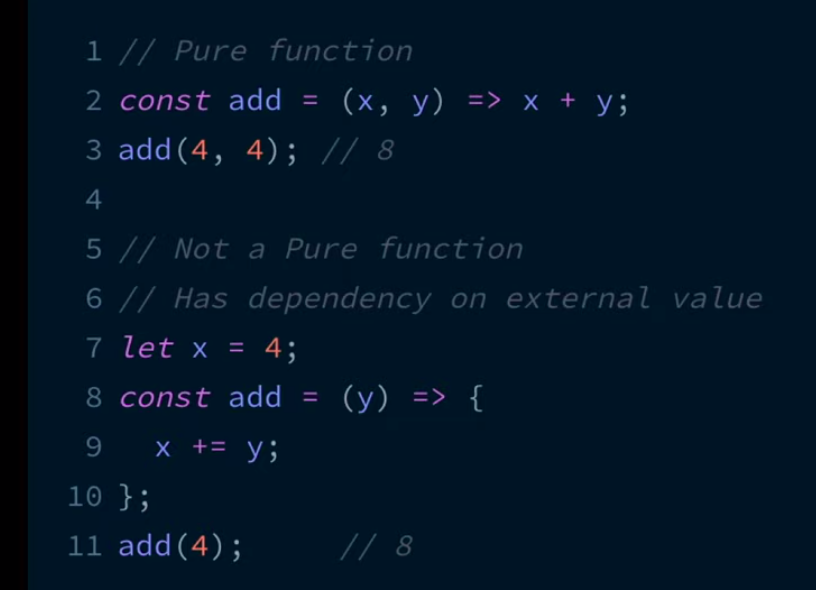
[Подробнее и больше методов](https://www.w3schools.com/js/js_string_methods.asp)

[**Методы массивов в JavaScript?**](https://youtu.be/CjdCxxqObaM?t=538)

<https://www.w3schools.com/js/js_array_methods.asp>

[Что такое чистая функция?](https://youtu.be/rlWgI7AvV18?t=401)

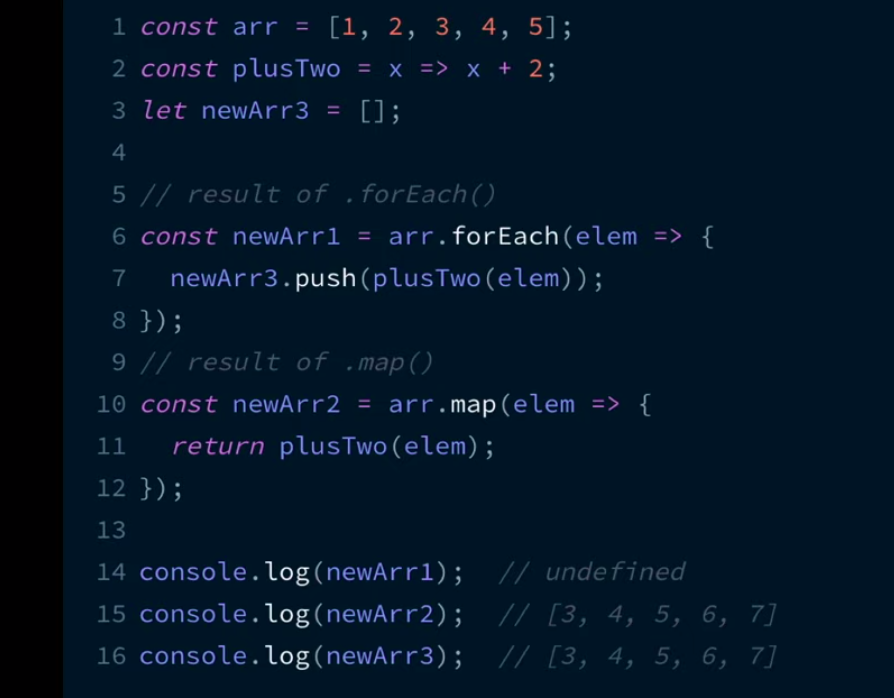
Чистая функция — это одна из концепций функционального программирования. Она должна удовлетворять двум условиям: 1) в ней не должно быть побочных эфектов; 2) каждый раз она возвращает одинаковый результат, когда вызывается с тем же набором аргументов. К побочным эффектам можно отнести видоизменение входных параметров, http, dom-запросы, изменения в файловой системе, а также вывод на экран.



[О функциональном программировании](https://doka.guide/js/fp/)

[Разница между .forEach() и .map()?](https://youtu.be/rlWgI7AvV18?t=456)

<https://www.w3schools.com/js/js_array_iteration.asp>



[Разница между .call(), .apply() и bind()?](https://youtu.be/rlWgI7AvV18?t=548)

[https://www.w3schools.com/js/js\_function\_call.asp](https://www.w3schools.com/js/js_function_call.asp)

[Статья](https://www.8host.com/blog/kak-rabotaet-this-bind-call-i-apply-v-javascript/)

[Еще статья](https://html-plus.in.ua/javascript-use-methods-bind-call-apply/)

[И еще](https://medium.com/@stasonmars/подробно-о-методах-apply-call-и-bind-необходимых-каждому-javascript-разработчику-ddd5f9b06290)

https://learn.javascript.ru/advanced-functions

#### 27. Для чего используется метод Function.prototype.apply?

Apply используется для привязки определенного объекта к значению this вызываемой функции.

**const** details = {

message: 'Hello World!'

}

**function** **getMessage**() {

**return** this.message

}

getMessage.apply(details) // Hello World!

Этот метод похож на Function.prototype.call. Единственное отличие состоит в том, что в apply аргументы передаются в виде массива.

**const** person = {

name: 'Marko Polo'

}

**function** **greeting**(greetingMessage) {

**return** `${greetingMessage} ${this.name}`

}

greeting.apply(person, ['Hello']) // Hello Marko Polo

#### 28. Для чего используется метод Function.prototype.call?

Call используется для привязки определенного объекта к значению this вызываемой функции.

**const** details = {

message: 'Hello World!'

};

**function** **getMessage**() {

**return** this.message;

}

getMessage.call(details); // Hello World!

Этот метод похож на Function.prototype.apply. Отличие состоит в том, что в call аргументы передаются через запятую.

**const** person = {

name: 'Marko Polo'

};

**function** **greeting**(greetingMessage) {

**return** `${greetingMessage} ${this.name}`;

}

greeting.call(person, 'Hello'); // Hello Marko Polo

#### 29. В чем разница между методами call и apply?

Отличие между call и apply состоит в том, как мы передаем аргументы в вызываемой функции. В apply аргументы передаются в виде массива, в call — через запятую.

**const** obj1 = {

result: 0

}

**const** obj2 = {

result: 0

}

**function** **reduceAdd**() {

**let** result = 0

**for** (**let** i = 0, len = arguments.length; i < len; i++) {

result += arguments[i]

}

this.result = result

}

reduceAdd.apply(obj1, [1, 2, 3, 4, 5]) // 15

reduceAdd.call(obj2, 1, 2, 3, 4, 5) // 15

#### 30. Для чего используется метод Function.prototype.bind?

Bind возвращает новую функцию, значением this которой является объект, указанный в качестве первого параметра. В отличие от bind, call и apply сразу же вызывают функцию.

**import** React **from** 'react'

**class** **MyComponent** **extends** **React**.**Component** {

**constructor**(props) {

super(props)

this.state = {

value: ''

}

this.handleChange = this.handleChange.bind(this)

// привязываем метод handleChange к компоненту MyComponent

}

**handleChange**(e) {

// код

}

**render**() {

**return** ( < >

<

input type = {

this.props.type

}

value = {

this.state.value

}

onChange = {

this.handleChange

}

/> </ >

)

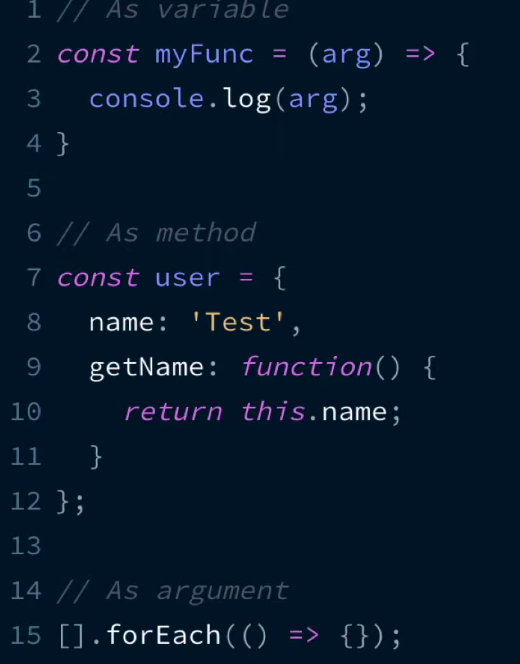
}

}

[**Почему в JS функции называют объектами первого класса?**](https://youtu.be/rlWgI7AvV18?t=624)

В JS любые типы данных ведут себя как объекты. Даже у примитивов есть набор методов, которые мы можем использовать.

Функции называют объектами первого класса, потому что они обрабатываются также, как и любое другое значение в JS. Они могут присваиваться переменным, быть свойством объекта (методом), элементом массива, аргументом другой функции, значением, возвращаемым функцией. Единственным отличием функции от любого другого значения в JS является то, что функция может быть выполнена или вызвана, и можно получить результат ее выполнения.



[**Как определить наличие свойства в объекте?**](https://youtu.be/kx3dR6ztICU?t=226)

Метод hasOwnProperty() возвращает логическое значение, указывающее, содержит ли объект указанное свойство.

## [Синтаксис](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty" \l "синтаксис)

obj.hasOwnProperty(prop)

### [Параметры](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty" \l "параметры)

prop

Имя проверяемого свойства.

## [Описание](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty" \l "описание)

Каждый объект, произошедший от [Object](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object), наследует метод hasOwnProperty. Этот метод может использоваться для определения того, содержит ли объект указанное свойство в качестве собственного свойства объекта; в отличие от оператора [in](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/in), этот метод не проверяет существование свойств в цепочке прототипов объекта.

## [Примеры](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty" \l "примеры)

### [Пример: использование](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty" \l "пример_использование_hasownproperty_для_проверки_существования_свойства)hasOwnProperty для проверки существования свойства

В следующем примере определяется, содержит ли объект o свойство с именем prop:

o = new Object();

o.prop = 'существует';

function changeO() {

o.newprop = o.prop;

delete o.prop;

}

o.hasOwnProperty('prop'); // вернёт true

changeO();

o.hasOwnProperty('prop'); // вернёт false

### [П](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty" \l "пример_собственные_и_унаследованные_свойства)ример: собственные и унаследованные свойства

Copy to Clipboard

Следующий пример показывает разницу между собственными свойствами и свойствами, унаследованными через цепочку прототипов:

o = new Object();

o.prop = 'существует';

o.hasOwnProperty('prop'); // вернёт true

o.hasOwnProperty('toString'); // вернёт false

o.hasOwnProperty('hasOwnProperty'); // вернёт false

### [П](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty" \l "пример_обход_свойств_объекта)ример: обход свойств объекта

Copy to Clipboard

Следующий пример показывает, как итерироваться по свойствам объекта с пропуском унаследованных свойств. Обратите внимание, что цикл [for...in](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/for...in) уже проходит только по перечисляемым элементам, так что не надо на основании отсутствия не перечисляемых свойств, показываемых в цикле, считать, что метод hasOwnProperty сам ограничивает свойства только перечисляемыми элементами (как это делает метод [Object.getOwnPropertyNames()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames)).

var buz = {

fog: 'stack'

};

for (var name in buz) {

if (buz.hasOwnProperty(name)) {

alert('это точно туман (' + name + '). Значение: ' + buz[name]);

}

else {

alert(name); // toString или что-то ещё

}

}

### [П](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty" \l "пример_hasownproperty_как_свойство)ример: hasOwnProperty как свойство

Copy to Clipboard

JavaScript не защищает имя свойства hasOwnProperty; таким образом, вполне может существовать объект с таким свойством, поэтому для получения правильного результата нужно использовать внешний метод hasOwnProperty:

var foo = {

hasOwnProperty: function() {

return false;

},

bar: 'Тут драконы'

};

foo.hasOwnProperty('bar'); // всегда возвращает false

// Используем метод hasOwnProperty другого объекта и вызываем его с передачей foo в качестве this

({}).hasOwnProperty.call(foo, 'bar'); // true

// Также для этих целей можно использовать свойство hasOwnProperty из прототипа Object

Object.prototype.hasOwnProperty.call(foo, 'bar'); // true

Обратите внимание, что в последнем случае новые объекты не создаются.

Copy to Clipboard



**Что такое IIFE?**

IIFE или Immediately Invoked Function Expression — это функция, которая вызывается или выполняется сразу же после создания или объявления. Для создания IIFE необходимо обернуть функцию в круглые скобки (оператор группировки), превратив ее в выражение, и затем вызвать ее с помощью еще одних круглых скобок. Это выглядит так: (function(){})().

(**function**( ) { }( ))

(**function**( ) { })( )

(**function** **named**(params) { })( )

(( ) => { })

(**function**(global) { })(window)

**const** utility = (**function**( ) {

**return** {

// утилиты

}

})

Все эти примеры являются валидными. Предпоследний пример показывает, что мы можем передавать параметры в IIFE. Последний пример показывает, что мы можем сохранить результат IIFE в переменной.  
  
Лучшее использование IIFE — это выполнение функций настройки инициализации и предотвращение конфликтов имен с другими переменными в глобальной области видимости (загрязнение глобального пространства имен). Приведем пример.

<script src="https://cdnurl.com/somelibrary.js"></script>

У нас есть ссылка на библиотеку somelibrary.js, которая предоставляет некоторые глобальные функции, которые мы можем использовать в нашем коде, но в этой библиотеке есть два метода, createGraph и drawGraph, которые мы не используем, потому что они содержат ошибки. И мы хотим реализовать эти функции самостоятельно.  
  
Одним из способов решить данную проблему является изменение структуры наших скриптов:

<script src="https://cdnurl.com/somelibrary.js"></script>

<**script**>

**function** **createGraph**() {

// код

}

**function** **drawGraph**() {

// код

}

</**script**>

Таким образом, мы переопределяем методы, предоставляемые библиотекой.  
  
Вторым способом является изменение имен наших функций:

<script src="https://cdnurl.com/somelibrary.js"></script>

<**script**>

**function** **myCreateGraph**() {

// код

}

**function** **myDrawGraph**() {

// код

}

</**script**>

Третий способ — использование IIFE:

<script>

**const** graphUtility = (**function**() {

**function** **createGraph**() {

// код

}

**function** **drawGraph**() {

// код

}

**return** {

createGraph,

drawGraph

}

})

</script>

В этом примере мы создаем служебную переменную, которая содержит результат IIFE, возвращающий объект, содержащий методы createGraph и drawGraph.  
  
Вот еще одна проблема, которую можно решить с помощью IIFE:

val li = document.querySelectorAll('.list-group > li')

**for** (**var** i - 0, len = li.length; i < len; i++) {

li[i].addEventListener('click', **function**(e) {

console.log(i)

})

}

Допустим, у нас есть элемент «ul» с классом «list-group», содержащий 5 дочерних элементов «li». И мы хотим выводить в консоль значение «i» при клике по отдельному «li». Однако вместо этого в консоль всегда выводится 5. Виной всему замыкание.  
  
Одним из решений является IIFE:

**var** li = document.querySelectorAll('.list-group > li')

**for** (**var** i = 0, len = li.length; i < len; i++) {

(**function**(currentIndex) {

li[currentIndex].addEventListener('click', **function**(e) {

console.log(currentIndex)

})

})(i)

}

[MDN Glossary](https://developer.mozilla.org/ru/docs/Glossary/IIFE)  
Причина, по которой этот код работает, как задумано, состоит в том, что IIFE создает новую область видимости на каждой итерации, и мы записываем значение «i» в currentIndex.

Такой паттерн раньше использовался для того, чтобы не допустить загрязнения глобального пространства имен. Применялся до появления модулей для создания поведения модуля.

[Metanit](https://metanit.com/web/javascript/3.3.php)

[**Что такое псевдомассив arguments?**](https://youtu.be/kx3dR6ztICU?t=442)

Коллекция аргументов, которые передаются в функцию. Псевдомассивом ее называют потому, что это объект подобный массиву. У него есть свойство length, но другие методы массивов ему недоступны. С помощью него можно получить доступ к любому аргументу, которые были переданы в функцию. В стрелочных функциях псевдомассив недоступен.

[](https://learn.javascript.ru/arguments-pseudoarray" \l ":~:text=Полный список аргументов%2C с которыми,методов массива у него нет.)More info

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/arguments

[**Разница между host-объектами и нативными объектами?**](https://youtu.be/kx3dR6ztICU?t=484)