**Frontend questions**

**JavaScript**

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

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

[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.

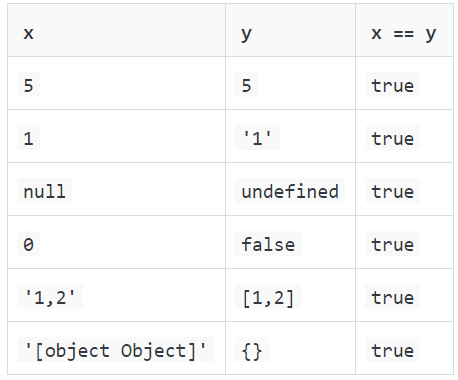
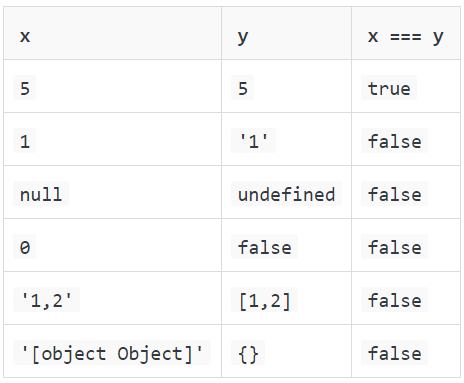
Оператор typeof позволяет нам увидеть, какой тип данных сохранён в переменной.

* 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](https://puzzleweb.ru/javascript/4_variables2.php" \l "a0) 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](https://puzzleweb.ru/javascript/13_functions3.php" \l "a1).
* Если функция вызывается как функция (а не как метод), ключевое слово [this](https://puzzleweb.ru/javascript/13_functions3.php" \l "a2) в качестве значения получает значение [undefined](https://puzzleweb.ru/javascript/5_types.php" \l "a9), 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:

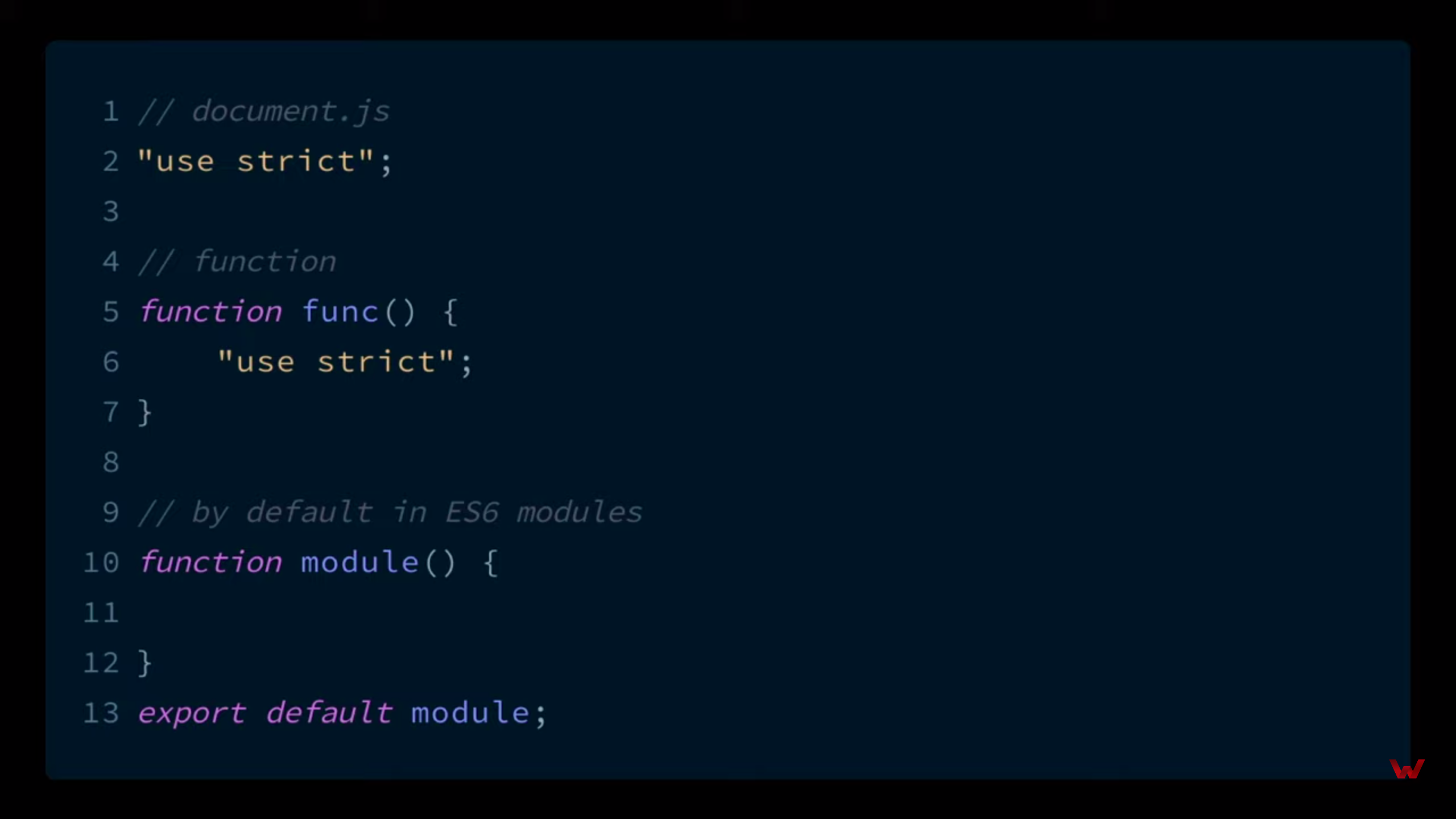
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | "use strict";  **function** foo() {  **var** a = 10;    eval("var x = 10; alert(x + a);");      alert(x);   // Ошибка. Такой переменной не существует  }    foo(); |

* 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) |

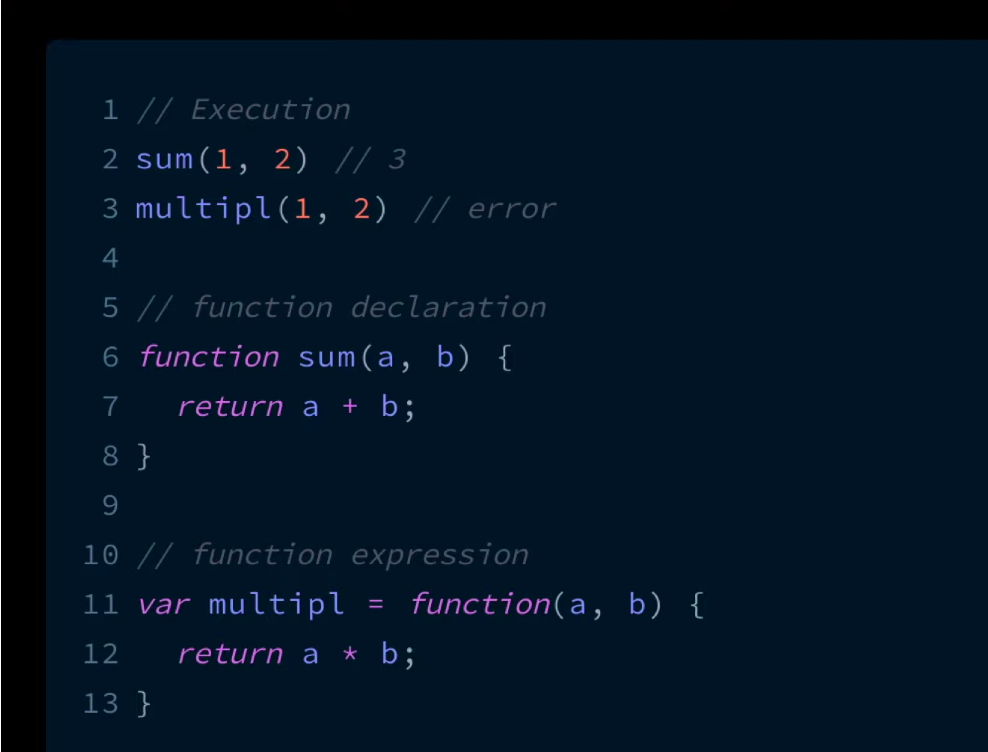
* Попытка присвоить значение свойству, недоступному для записи, или создание нового свойства в нерасширяемом объекте порождают исключение.
* Обращение к свойствам arguments.caller и arguments.callee порождает исключение.
* Функции, объявленные как FD внутри блока, находятся в блочной области видимости:

|  |  |
| --- | --- |
| 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.