**Привет, мир!**

**Тег "script"**

Для того, чтобы подключить JS к веб-странице, необходимо разместить его через тег <script>

Есть два варианта его использования:

* Прямо внутри html
* В отдельном файле
  + Польза отдельных файлов, в том, что браузер их кэширует
  + Соответственно если на другой странице будет тот же скрипт - он достанет его из кэша

Если аттрибут src установлен, то содержимое внутри <script> игнорируется

* Для добавления кода JavaScript на страницу используется тег <script>
* Атрибуты type и language необязательны.
* Скрипт во внешнем файле можно вставить с помощью <script src="path/to/script.js"></script>.

**Исполнение скрипта через Node.js**

Чтобы выполнить скрипт через Node.js необходимо выполнить команду в терминале. Например, чтобы выполнить скрипт с именем script.js:

node script.js

**Структура кода**

**Инструкции**

Инструкция - это синтаксическая конструкция или команда, которая выполняет некое действие. Например alert('Привет, мир!') - это инструкция.

**Точка с запятой**

Для разделения инструкций используется ; - точка с запятой.

Либо же новая строка, тогда движок пользуется правилами [автоматической вставки точки с запятой](https://tc39.es/ecma262/#sec-automatic-semicolon-insertion) описанной в спецификации ECMA-262

**Комментарии**

JS поддерживает два вида комментариев

* Однострочные комментарии

// Я комментарий

* Многострочные комментарии

/\*

Я комментарий

Многострочный

\*/

Многострочные комментарии используются в [JSDoc](https://jsdoc.app/) при описании документации к коду.

**Строгий режим**

Директива выглядит как строка: "use strict" или 'use strict'. Когда она находится в начале скрипта, весь сценарий работает в «современном» режиме.

"use strict";

// этот код работает в современном режиме

// ...

Современный JavaScript поддерживает «классы» и «модули», в них строгий режим включен по-умолчанию.

**Что дает**

* нельзя присваивать значение в неопределённую переменную
* нельзя использовать инструкцию with
* в ES5 нельзя определить повторные свойства в литерале объекта
* нельзя определить повторные формальные параметры функции
* изменения объекта arguments не изменяют аргументы
* delete приводит к ошибке, если аргумент - не изменяемое свойство объекта
* eval не может инстанциировать переменные и функции в контексте вызова
* this не преобразуется в объект, а если значение this - undefined или null, то не преобразуется в глобальный объект
* eval и arguments - нельзя изменить или использовать в качестве имени
* нельзя использовать argument.caller и arguments.callee
* больше слов, зарезервированных для использования в будущем
* нельзя использовать литералы восьмеричной СС

Переменные используются для хранения некой информации. Из названия ясно, что переменные могут... меняться. Переменные в JS хранятся в памяти браузерного процесса, проще говоря в оперативной памяти.

## Объявление переменных

В JavaScript переменные можно объявить при помощи ключевых слов let или при помощи var. Между ними есть разница, которая будет разобрана позже, пока нужно просто понимать, что лучше использовать современный вариант let

Имена переменных:

* Имя переменной должно содержать только буквы, цифры или символы $ и \_
* Первый символ не должен быть цифрой

Заметки:

* Переменные объявляют в camelCase.
* Регистр имеет значение. apple и Apple - разные переменные
* Нелатинские буквы разрешены, но не рекомендуются
* Имеется [список зарезервированных слов](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Lexical_grammar#%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%B2%D1%8B%D0%B5_%D1%81%D0%BB%D0%BE%D0%B2%D0%B0), которыми нельзя назвать переменную

Константы объявляются через ключевое слово const. Используется, если переменная не должна меняться в ходе выполнения.

Именуем в uppercase, если константа известна до выполнения кода

Именуем в camelCase, если константа вычисляется во время выполнения кода

## Управление памятью. Введение

[Подробнее](https://tproger.ru/translations/upravlenie-pamjatju-v-javascript/)

Каждый раз, когда мы объявляем переменную или создаем функцию, необходимая для этого память проходит следующие этапы:

* Выделение
  + JS выделяет столько памяти, сколько нам понадобится для конкретной переменной.
* Использование
  + Чтение и запись в переменную
* Освобождение
  + Как только переменная перестает быть нужной - освобождается память под другие процессы.

## Стек и куча

**Стек** (англ. stack) - это структура данных, которая используется для хранения статических данных. В JS сюда попадают примитивные значения: undefined, null, string, number, boolean, symbol) + ссылки на функции и объекты

* Используется для примитивных значений и ссылок
* Размер известен во время компиляции
* Выделяется фиксированный объем памяти

**Куча** (англ. memory heap) - динамическое выделение памяти. Используется для хранения объектов и функций.

* Используется для объектов и функций
* Размер становится известен во время выполнения
* Объем не ограничен.

Пример:

// Объект, хранится в куче

// В куче создается объект, а в стеке хранится ссылка на него

const person = {

name: 'John', // Примитив, хранится в стеке

age: 24, // Примитив, хранится в стеке

};

// Массив, тоже объект, хранится в куче

const hobbies = ['hiking', 'reading'];

let name = 'John'; // выделяет память для строки

const age = 24; // выделяет память для числа

name = 'John Doe'; // выделяет память для новой строки

const firstName = name.slice(0,4); // выделяет память для новой строки

// Примитивные значения по своей природе иммутабельны: вместо того, чтобы изменить начальное значение,

// JavaScript создает еще одно.

# Типы данных

JS является динамически типизированным языком. Это значит, что в переменных может быть в один момент времени, например, строка, а в другой момент времени - число.

На данный момент в JS есть 8 основных типов данных:

* undefined
  + Означает «значение не было присвоено».
  + для проверок, была ли переменная назначена.
  + typeof вернет 'undefined'
* null
  + Означает «ничего», «пусто» или «значение неизвестно».
  + для проверок, является ли переменная пустой
  + typeof вернет 'object'.
    - Это баг, раньше в первых версиях JS, значения представлялись как type tag и value.
    - У объекта type tag был 0.
    - null был представлен как нулевой указатель, соответственно тоже стал иметь type tag = 0
* string
  + для строк. Строка может содержать ноль или больше символов, нет отдельного символьного типа.
* boolean
  + для true/false
* number
  + Специальные числовые значения:
    - Infinity - математическая бесконечность. Больше любого числа
    - -Infinity - отрицательная математическая бесконечность. Меньше любого числа
    - NaN - вычислительная ошибка.
      * Любая операция с NaN возвращает NaN
      * NaN не равен NaN
* bigint
  + Для целых чисел произвольной длины
  + В JS number не может содержать в себе числа больше 2^53, или меньше чем -(2^53 - 1)
  + Позволяет работать с числами произвольной длины
* symbol
  + Для уникальных идентификаторов
* object
  + Для сложных структур данных

# Взаимодействие: alert, prompt, confirm

* alert
  + Показывает сообщение и ждет пока пользователь нажмет "ОК"
  + Возвращает undefined
* prompt
  + Отображает модальное окно с текстом и полем для ввода
  + Возвращает string, текст введенный в поле пользователем или null если ввод отменен
  + Для IE следует указывать значение по-умолчанию
* confirm
  + Отображает модальное окно с текстом и двумя кнопками "ОК" и "Отмена".
  + Возвращает boolean

Данные методы являются "модальными" и останавливают работу скрипта, а также не позволяют пользователю взаимодействовать с остальной частью страницы.

Также данные окна нельзя кастомизировать.

# Преобразование типов

Это процесс конвертации значения из одного типа в другой. В JS существует всего 3 типа конвертации:

* Строковое
* Численное
* Булевое

Существуют явные и неявные преобразования к типу:

* Явное - при применении функций String(), Number(), Boolean()
* Неявное - при вызове операторов

Полезные ресурсы:

* [JavaScript Comparison Table](https://dorey.github.io/JavaScript-Equality-Table/)
* [wtfjs.com](https://wtfjs.com/)

## Строковое преобразование

Как преобразовать:

* String(value)
* ${value}
* value.toString()

Примеры:

// Объекты преобразуются через Symbol.toPrimitive или методы toString/valueOf

String({ a: 1 }) // '[object Object]'

String([1,2,3]) // '1,2,3'

// Примитивы преобразуются к строке очевидным образом

String(undefined) // 'undefined'

String(1) // '1'

String(null) // 'null'

String(true) // 'true'

String(false) // 'false'

String(Symbol('123')) // 'Symbol(123)'

String(NaN) // 'NaN'

String(Infinity) // 'Infinity'

String(-Infinity) // '-Infinity'

Интересно, что символы преобразуются к строке только явно:

String(Symbol('my symbol')) // 'Symbol(my symbol)'

'' + Symbol('my symbol') // TypeError is thrown

### Численное преобразование

Как преобразовать:

* Операторы сравнения. >, <, <=, >=
* Бинарные операторы. |, &, ^, ~
* Арифметические операторы. \*, -, /, %, \*\*
  + Особняком стоит +
    - Если один из операндов является строкой, преобразует оба к строке
* Унарный оператор +
* Оператор нестрого равенства ==, !=
  + Не вызывает численное преобразование, если оба операнда являются строками

Пример неявного преобразования:

2 + 2 + '2' // 42, сначала складываются первые два числа, затем конкатенация

При преобразовании строки в число, движок сначала отсекает все пробелы слева и справа, затем пытается преобразовать. Вернет либо число, либо NaN

Number(' 12 ') // 12

Number('anything') // NaN

Примеры преобразований:

// Примитивы

Number(undefined) // NaN

Number(null) // 0

Number('1') // 1

Number('anything') // NaN

Number(Infinity) // Infinity

Number(-Infinity) // -Infinity

Number(Symbol()) // Typeerror: Cannot convert a Symbol value to a number

// Объекты

Number([]) // 0

Number([10]) // 10

Number([10, 15]) // NaN

Number({}) // NaN

Number(new Date()) // 1641093339094, unix timestamp

Есть **два** специальных правила:

* При применении == к null или undefined численного преобразования не происходит.

null == 0 // false, null не конвертируется в 0

null == null // true

undefined == undefined // true

null == undefined // true

* NaN не равен ничему, даже самому себе.

NaN == NaN // false

NaN === NaN // false

## Булевое преобразование

Как преобразовать:

* Явно. Boolean(2)
* Неявно. Через логический контекст. if (2) { ... }
* Неявно. Через логический оператор. !!2
* Неявно. Через логический оператор. 2 || 'hello'

Логические операторы такие как || и && производят булевое преобразование под капотом, но при этом всегда возвращают оригинальное значение операндов, даже если они не являются булевыми.

let x = 'hello' && 123; // x = 123

let y = 'hello' || 123 // y = 'hello'

Все falsy-значения:

Boolean('') // false

Boolean(0) // false

Boolean(-0) // false

Boolean(NaN) // false

Boolean(null) // false

Boolean(undefined) // false

Boolean(false) // false

Остальные значения будут true

# Операторы

[Полный список операторов](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#logical_not)

## Термины

* Операнд (аргумент) - то, к чему применяется оператор
  + 5 \* 2, 5 и 2 это операнды
* Унарный оператор - оператор, который применяется только к одному операнду
  + -5, тут - является унарным оператором, так как применяется только к операнду 5
* Бинарный оператор - оператор, который применяется к двум операндам.
  + 5 - 2, тут - - является бинарным оператором, так как применяется к операндам 5 и 2

## Возведение в степень \*\*

Оператор работает и для нецелых чисел. Например, квадратный корень является возведением в степень 1/2

4 \*\* (1/2) // 2 (степень 1/2 эквивалентна взятию квадратного корня)

8 \*\* (1/3) // 2 (степень 1/3 эквивалентна взятию кубического корня)

## Сложение строк

Если применить к строкам оператор + - это будет конкатенация.

Если хотя бы один из операндов является строкой - для другого операнда будет произведено строковое преобразование, а затем будет конкатенация из двух операндов.

## Приведение к числу, унарный +

Унарный плюс - это аналог Number(value), только более короткий

Пример:

+' 24 ' // 24

# Операторы сравнения

Речь пойдет об операторах >, >=, <, <=, ==, ===.

Результат сравнения всегда имеет логический тип, т.е возвращает true/false

## Сравнение строк

Для определения, что одна строка больше другой JS использует "алфавитный" или "лексикографический" порядок. В общем, посимвольное сравнение.

JS сравнивает не сами символы, а их значение из Unicode-таблицы.

## Сравнение разных типов

При сравнении различных типов, JS приводит каждый операнд к числу. Исключением является сравнение с помощью операторов строгого равенства/неравенства.

## Сравнение с null и undefined

Значения null и undefined равны == друг другу и не равны любому другому значению.

#### При строгом равенстве ===:

null === undefined // false

#### При нестрогом равенстве ==:

null == undefined // true

#### При использовании математических операторов и других операторов сравнения:

Преобразуются к числам, null становится 0, а undefined - NaN

# Условное ветвление: if, '?'

Используется для выполнения различных инструкций в зависимости от условий.

## Инструкция "if"

1. Вычисляет условие в скобках
2. Если результат true - выполняет блок кода

if (age >= 18) console.log('Your age more or equal 18')

if (age < 18) {

alert('You are too young!')

console.log('Your age less than 18')

}

## Преобразование к логическому типу

После вычисления условия в скобках, результат преобразуется к логическому типу.

## Блок "else"

Выполняется, если условие в "if" ложно.

## Несколько условий "else if"

Проверяет дополнительное условие и выполняет блок кода, если true

## Условный (тернарный) оператор

let result = условие ? значение1 : значение2;

Сначала вычисляется условие: если оно истинно, тогда возвращается значение1, в противном случае – значение2.

# Логические операторы

Логические операторы в JS это

* ||, логическое ИЛИ (сложение), дизъюнкция
* &&, логическое И (умножение), конъюнкция
* !, логическое отрицание, инверсия

## Логическое ИЛИ (||)

Возвращает true, если хотя бы один из операндов является true.

Если значение не логического типа, то оно к нему приводится в целях вычислений.

* Находит первое **истинное** значение
  + Вычисляет операнды слева направо.
  + Каждый операнд конвертирует в логическое значение. Если результат true, останавливается и возвращает исходное значение этого операнда
  + Если все операнды являются ложными, возвращает последний.

## Логическое И (&&)

Возвращает true, если оба операнда являются true

* Находит первое **ложное** значение
  + Вычисляет операнды слева направо.
  + Каждый операнд конвертирует в логическое значение. Если результат false, останавливается и возвращает исходное значение этого операнда
  + Если все операнды были истинными, возвращает последний.

Не стоит забывать, что приоритет у && больше, чем у ||

## Логическое НЕ (!)

Оператор принимает один операнд, т.е является унарным, и выполняет следующие действия:

1. Сначала приводит аргумент к логическому типу true/false
2. Затем возвращает противоположное значение

Часто используется, чтобы привести переменную к логическому типу, двойное отрицание:

let x = 0

!x // true

!!x // false

# Оператор слияния с null '??'

Оператор '??':

* Возвращает первый операнд, если он не является null или undefined
* Возвращает второй операнд, если первый является null или undefined

Этот оператор является аналогом вот такой записи:

result = (a !== null && a !== undefined) ? a : b;

## Сравнение с ||

Ранее, в JS для такой проверки использовался || оператор. Но его проблема в том, что он пропускает все falsy-значения

Т.е например если мы решим проверить:

let height = 0;

alert(height || 100); // 100, 0 - это falsy значение, поэтому возвращает 100

alert(height ?? 100); // 0, 0 - это не null/undefined, поэтому возвращает 0

* || возвращает первое истинное значение.
* ?? возвращает первое определённое значение.

## Приоритет

Не стоит забывать, что ?? имеет низкий приоритет 5.

let height = null;

let width = null;

// важно: используйте круглые скобки

let area = (height ?? 100) \* (width ?? 50);

alert(area); // 5000

Или же, если без скобок, то движок будет считывать это так:

// без круглых скобок

let area = height ?? 100 \* width ?? 50;

// ...то же самое, что предыдущее выражение (вероятно, это не то, что нам нужно):

let area = height ?? (100 \* width) ?? 50;

## Использование с && или ||

JS ограничивает использование оператора ?? вместе с && или ||.

let x = 1 && 2 ?? 3; // Синтаксическая ошибка

Чтобы избежать ошибки, надо использовать группировку.

let x = (1 && 2) ?? 3; // Работает без ошибок

alert(x); // 2

# Циклы while и for

Циклы предназначены для многократного повторения кода. Одно выполнение цикла называется "итерация".

## while

let i = 0;

while (i < 3) { // выводит 0, затем 1, затем 2

alert( i );

i++;

}

Выполняет блок кода, пока действует условие.

## do ... while

do {

// тело цикла

} while (condition);

Сначала выполнит блок кода, затем проверит условие. И если оно истинно - будет выполнять блок кода снова и снова.

Нужен, когда необходимо, чтобы тело выполнилось **хотя бы один раз**

## for

for (начало; условие; шаг) {

// ... тело цикла ...

}

* начало; выполняется один раз при входе в цикл
* условие; проверяется перед каждой итерацией. Если вычислится в false - цикл остановится
* шаг; выполняется после тела цикла на каждой итерации перед проверкой условия

#### Полезное:

* Любая часть for может быть пропущена
* Переменные существуют только внутри цикла и ограничены своим scope'ом

#### Конструкции внутри цикла:

* break - выход из цикла
* continue - переход к следующей итерации цикла (вроде break, только прерывает итерацию, а не цикл)

Нельзя использовать break/continue справа от оператора „?“

#### Метки break/continue

Полезно при вложенных циклах, когда в дочернем цикле надо выйти из родительского.

outer: for (...) {

...

}

Вызов break/continue возможен только внутри цикла, и метка должна находиться где-то выше этой директивы.

Пример использования:

outer: for (let i = 0; i < 3; i++) {

inner: for (let j = 0; j < 3; j++) {

if (i === j) continue outer

if (i + j >= 4) break outer

console.log(i, j)

}

}

# Конструкция "switch"

Заменяет собой сразу несколько if

switch(x) {

case 'value1': // if (x === 'value1')

...

[break]

case 'value2': // if (x === 'value2')

...

[break]

default:

...

[break]

}

Этапы:

* Переменная x проверяется на строгое равенство первому значению value1, затем второму value2 и так далее.
* Если соответствие установлено – switch начинает выполняться от соответствующей директивы case и далее, до ближайшего break (или до конца switch).
* Если ни один case не совпал – выполняется (если есть) вариант default

Проверка всегда строгая, поэтому тип имеет значение.

#### Если break нет, то выполнение пойдёт ниже по следующим case, при этом остальные проверки игнорируются.

Так можно группировать кейсы

let a = 2 + 2;

switch (a) {

case 4:

alert('Правильно!');

break;

case 3: // (\*) группируем оба case

case 5:

alert('Неправильно!');

alert("Может вам посетить урок математики?");

break;

default:

alert('Результат выглядит странновато. Честно.');

}

#### Любое выражение может быть аргументом для switch/case

let a = "1";

let b = 0;

switch (+a) {

case b + 1:

alert("Выполнится, т.к. значением +a будет 1, что в точности равно b+1");

break;

default:

alert("Это не выполнится");

}

# Функции (Function Declaration)

Функции являются основными "строительными блоками" программы. Используются для того, чтобы не повторять один и тот же код в разных частях программы.

JavaScript позволяет создавать функцию различными способами:

* Function Declaration;
* Function Expression;
* Arrow Function.

## Объявление функции

function showMessage() {

alert( 'Всем привет!' );

}

При объявлении функции через Function declaration - можно ее объявить даже в конце кода, а пользоваться в начале, так как интерпретатор перед запуском скрипта проходится и собирает все Function declaration, переменные и классы.

Называется такое понятия Hoisting (поднятие).

#### Приоритеты hoisting

Объявления функций «поднимаются» над объявлением переменных, но не над их назначениями.

##### Назначение переменной над объявлением функции.

var double = 22;

function double(num) {

return (num\*2);

}

console.log(typeof double); // Вывод: number

##### Объявление функции над объявлением переменной.

var double;

function double(num) {

return (num\*2);

}

console.log(typeof double); // Вывод: function

Используя es5 переменную var, попытки использования необъявленных переменных приведут к тому, что переменной будет назначено значение undefined при «поднятии»

## Область видимости

* Переменные внутри функции видны только внутри функции
* Функция имеет доступ к переменным, которые находятся выше по области видимости
* Нельзя извне получить локальные переменные функции
* Если название переменной внутри функции и снаружи совпадает - берется локальная переменная

## Аргументы функции

arguments - массиво-подобный объект, локальная переменная, которая содержится в каждой функции, кроме стрелочных.

Передаваемые значения копируются в параметры функции и становятся локальными переменными. Т.е. если внутри функции поменяем значение аргумента, то оно не будет видно снаружи.

Но если значение является ссылкой, то его изменения будет видно за пределами функции

Таким образом, **функция не может мутировать примитивные типы, но может мутировать объекты**

## Возврат значения

Производится с помощью return. Если return не указан, или он пустой - функция вернет undefined. После return функция останавливается (вроде break в циклах)

Не стоит добавлять пустую строку после return, иначе интерпретатор поставит точку с запятой после return Поэтому функция вернет undefined

# Function Expression

По сути без разницы, как мы определили функцию, это просто **значение (ссылка)**, хранимое в переменной.

#### Function declaration

function pow(x, n) {

return x \*\* n

}

#### Function expression

const pow = function(x, n) {

return x \*\* n

}

#### Arrow function

const pow = (x, n) => x \*\* n

## Что такое callback

Простыми словами: callback — это функция, которая должна быть выполнена после того, как другая функция завершила выполнение (отсюда и название: callback – функция обратного вызова).

В JS функции — это объекты. Поэтому функции могут принимать другие функции в качестве аргументов, а также функции могут возвращать функции в качестве результата.

Функции, которые это умеют, называются функциями высшего порядка. А любая функция, которая передается как аргумент, называется callback-функцией.

## Отличия Function Expression от Function Declaration

* Синтаксис
* Hoisting. Function Declaration можно использовать во всем скрипте (или блоке кода, если функция объявлена в блоке)
* В строгом режиме, когда Function Declaration находится в блоке {...}, функция доступна везде внутри блока. Но не снаружи него.

# Стрелочные функции, основы

Существует третий синтаксис объявления функции, кроме Function expression. Это стрелочные функции.

let func = (arg1, arg2, ...argN) => expression

* Если у нас только один аргумент, скобки можно опустить
* Если нет аргументов, указываются пустые скобки
* Если функция в одну строку - возвращает результат без return
* Если многострочная, нужно использовать return как обычно

## Отличия стрелочной функции от обычной

[Полезная статья об отличиях](https://dmitripavlutin.com/differences-between-arrow-and-regular-functions/)

* Не определяет свой контекст
* Не может быть использована как конструктор
* Не имеет переменной arguments
* Не может быть использована в качестве генератора
* Стрелочная в одну строку позволяет не использовать return
* Если использовать метод в классе
  + Обычную функцию придется биндить при передаче коллбэком
  + Стрелочную не обязательно биндить

# Отладка в браузере Chrome

Приостановить выполнение скрипта можно тремя способами:

* Точками останова.
* Использованием в коде команды debugger.
* При ошибке (если инструменты разработчика открыты и опция включена).

[Песочница](https://learn.javascript.ru/article/debugging-chrome/debugging/index.html%D0%BF)

## Точки останова

[Официальная документация Google chrome devtools](https://developer.chrome.com/docs/devtools)

Можно поставить 3 разных вида точек останова:

* Обычный брейкпоинт
* Условный брейкпоинт (остановит только при совершении условия)
* Логпоинт (выведет консоль лог)

1. Watch - показывает текущие значения выражений
2. Call Stack показывает последовательность вызовов функций
3. Scope показывает текущие переменные.

#### Continue to here

Позволяет продвинуться на несколько шагов до нужной точки, минуя брейкпоинты

# Советы по стилю кода

Вообще - чтобы довольно легко и комфортно изучить правила написания кода - надо поставить себе линтер. А дальше просто писать код - и со временем вы привыкнете писать так, как говорит линтер.

Стоит рассмотреть [Standard JS](https://standardjs.com/)

Когда мы думаем о написании «лучшего» кода, мы должны задать себе вопросы: «Что сделает код более читаемым и лёгким для понимания?» и «Что может помочь избегать ошибок?». Это – основные моменты, о которых следует помнить при выборе и обсуждении стилей кода.

## Популярные стайл-гайды

* [Google JavaScript Style Guide](https://google.github.io/styleguide/javascriptguide.xml)
* [Airbnb JavaScript Style Guide (есть перевод)](https://github.com/airbnb/javascript)
* [Idiomatic.JS (есть перевод)](https://github.com/rwaldron/idiomatic.js)
* [StandardJS](https://standardjs.com/)

## Кастомная настройка линтера

ESLint - сейчас практически "золотой стандарт". Встречается почти во всех проектах, легко настраивается под себя.

# Комментарии

## Хорошие комментарии

* Общая архитектура
* Использование функций, их параметры. JSDoc в помощь
* Неочевидные решения, важные детали

## Плохие комментарии

* Комментарии, которые поэтапно объясняют как работает код. Код должен быть **самодокументируемым**
  + Если код не ясен, лучше попробовать его переписать и разделить на более логичные модули

# Ниндзя-код

* Краткость, сестра таланта
  + Не стоит пытаться написать максимально коротко - надо писать максимально понятно
* Именование переменных
  + Нужно именовать переменные правильно и очевидно. Чтобы было ясно что они делают/хранят
  + Не нужно просто так сокращать переменные
  + Имя переменной не должно быть абстрактным
  + Не стоит использовать рядом похожие имена переменных
  + На траслите переменные тоже не стоит именовать...
  + Никаких mega, super и т.д. в имени переменной
* Не стоит перекрывать внешние переменные внутри функций
* Функции должны быть максимально чистыми
* Функции/методы должны делать что-то одно.

# Автоматическое тестирование c использованием фреймворка Mocha

Речь об автоматическом тестировании.

## Зачем нужны тесты

* При тестировании кода ручными перезапусками легко упустить что-нибудь важное.
* Код, хорошо покрытый тестами, как правило, имеет лучшую архитектуру.

В BDD сначала пишут спецификацию, а потом реализацию. В конце у нас есть и то, и другое.

Спецификация используется в качестве:

1. Пример. Тесты являются своего рода примерами использования функций/классов и ожидаемого поведение.
2. Тест. Тесты позволяют рефакторить код без страха сломать предыдущее поведение.
3. Документация. В спецификации человеческим языком написано, что должна выполнять функция.

## Понятия:

* [BDD](https://ru.wikipedia.org/wiki/BDD_(%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))
  + Разработка через поведение. Ответвление от TDD
  + Описание желаемого поведения даётся с помощью спецификации поведения
* [TDD](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%87%D0%B5%D1%80%D0%B5%D0%B7_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
  + Разработка через тестирование

## Процесс разработки

1. Пишется начальная спецификация с тестами, проверяющими основную функциональность.
2. Создается начальная реализация.
3. Пишем функцию, пока все тесты не будут зелеными.
4. Дорабатываем тесты
5. Рефакторим код.
6. Повторяем шаги, пока не будет готово.

Один тест проверяет одну вещь.

# Полифиллы

JS - динамично-развивающийся язык, в котором постоянно появляются какие-то новые фичи. Сначала фичи публикуются в [черновике](https://tc39.github.io/ecma262/), затем разработчики языка решают какую из фич лучше реализовать в первую очередь и публикуют их в [спецификации](http://www.ecma-international.org/publications/standards/Ecma-262.htm).

Таким образом, довольно часто реализуются только часть стандарта.

[Таблица поддержки различных фич](https://kangax.github.io/compat-table/es6/)

## Babel

Babel - это транспайлер. Переписывает современный JS код с предыдущий стандарт.

Babel состоит из двух частей:

1. Переписывает код в старый стандарт.
2. Добавляет полифиллы (функционал, которого нет в текущем стандарте. Добавляет свои реализации).

# Объекты

Объекты используются для хранения коллекций различных значений и более сложных сущностей.

## Объявление

let user = new Object(); // синтаксис "конструктор объекта"

let user = {}; // синтаксис "литерал объекта"

* Ключ, имя, идентификатор, свойство.
* Значение, значение свойства.

Для удаления свойства можно использовать оператор delete

## Вычисляемые свойства

let fruit = prompt("Какой фрукт купить?", "apple");

let bag = {

[fruit]: 5, // имя свойства будет взято из переменной fruit

};

## Ограничения на имена свойств

Для объекта нет ограничений. Имена свойств могут быть **ИЛИ СТРОКАМИ ИЛИ СИМВОЛАМИ**. Все остальные типы данных будут преобразованы к строке.

Есть особенность, связанная со свойством \_\_proto\_\_. Его нельзя установить в необъектное значение. Присвоение примитивного значения будет просто игнорироваться.

## Проверка существования свойства, оператор «in»

let obj = {

test: undefined

};

alert( obj.test ); // выведет undefined, значит свойство не существует?

alert( "test" in obj ); // true, свойство существует!

## Цикл for ... in

Выполняется для каждого **СВОЙСТВА** объекта. Т.е для каждого **КЛЮЧА**.

let user = {

name: "John",

age: 30,

isAdmin: true

};

for (let key in user) {

// ключи

alert( key ); // name, age, isAdmin

// значения ключей

alert( user[key] ); // John, 30, true

}

## Упорядочение свойств объекта

Свойства упорядочены особенным образом. Свойства с целочисленными ключами сортируются по возрастанию, остальные располагаются в порядке создания.

Сначала идут отсортированные целочисленные ключи, затем остальные в порядке создания.

Целочисленное свойство - это integer, который может быть преобразован из строки в number и обратно без изменений

* "49" - целочисленный
* "-49" - не целочисленное
* "+49" - не целочисленное
* "49.2" - не целочисленное

# Копирование объектов и ссылки

Примитивные типы: строки, числа, логические значения - присваиваются и копируются "по значению".

Объекты ведут себя иначе. Они хранят не значение, а ссылку на объект.

## Сравнение по ссылке

Два объекта равны только в том случае, если у них совпадает ссылка. Т.е они ссылаются на один и тот же объект.

## Копирование и объединение объектов

* Можно через Object.assign
* Можно через spread syntax (...)

Важно понимать, что это не глубокое копирование.

Если надо глубокое - либо взять из лодаша функцию, либо написать свою.

Еще есть хак с JSON.parse(JSON.stringify(variable\_name)) - но он потеряет прототип и методы.

# Сборка мусора

Основной концепцией управления памятью в JavaScript является принцип достижимости.

Достижимые значения - это те, что "достижимы" и используются. Они гарантированно находятся в памяти.

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

* Локальные переменные и параметры текущей функции
* Переменные и параметры других функций в текущей цепочке вызовов.
* Глобальные переменные.

Эти значения можно называть корнями

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

Он следит за всеми объектами и удаляет те, которые стали недостижимы.

## Внутренние алгоритмы

Основной алгоритм сборки мусора – «алгоритм пометок» (англ. **«mark-and-sweep»**)

* Сборщик помечает все корневые объекты
* Затем идет по ссылкам и помечает все найденные объекты
* Затем идет по ссылкам помеченных объектов.
  + Все объекты запоминаются, чтобы в будущем не посещать один и тот же объект дважды
* И так далее, пока не будут помечены все объекты от корня
* Непомеченные объекты - удаляются

#### Некоторые из оптимизаций

* Сборка по поколениям (**Generational collection**)
  + Объекты делятся на "новые" и "старые". Те, которые живут достаточно "долго" - становятся "старыми" и реже проверяются оптимизатором
* Инкрементальная сборка (**Incremental collection**)
  + Интерпретатор при большом кол-ве объектов пытается организовать сборку мусора поэтапно
* Сборка в свободное время (**Idle-time collection**)
  + Интерпретатор старается запускать сборку только во время простоя процессора.

# Методы объекта, this

Функция, которая является свойством объекта называют **методом**

Есть три варианта присвоения:

1. Обычный

user = {

sayHi: function() {

alert('Привет');

}

}

1. Сокращенный

user = {

sayHi() {

alert('Привет');

}

}

1. Присвоение после объявления объекта

user = {

name: 'Maxim'

}

user.sayHi = function() {

alert('Привет')

}

## Ключевое слово "this" в методах

Для доступа к информации внутри объекта метод может использовать ключевое слово this

Значение this - это объект "перед точкой", который использовался для вызова метода.

## Вызов без объекта: this == undefined

function sayHi() {

alert(this);

}

sayHi(); // undefined

* В строгом режиме - this вернет undefined. Любое обращение к свойству выбросит ошибку
* В нестрогом режиме - this вернет window.

## Свободный this

В JavaScript this является "свободным". Т.е его значение вычисляется в момент вызова метода:

* Не зависит от того, где этот метод был объявлен
* Зависит от того, какой объект вызывает метод (какой объект "стоит перед точкой")

## Внутренняя реализация: ссылочный тип

Для работы вызовов типа user.hi() , JavaScript использует трюк – точка '.' возвращает не саму функцию, а специальное значение «ссылочного типа», называемого Reference Type.

Reference Type - это внутренний тип. Под капотом он выглядит вроде этого:

(base, name, strict)

// (user, "hi", true)

где:

* base - это объект
* name - это имя свойства объекта
* strict - это режим исполнения (use strict)

Результатом доступа к user.hi является не функция, а значение ссылочного типа.

При любой другой операции, значение ссылочного типа заменяется на обычную функцию.

Таким образом, значение this передаётся правильно, только если функция вызывается напрямую с использованием синтаксиса точки obj.method() или квадратных скобок obj['method']() (они делают то же самое).

# Конструкторы, создание объектов через "new"

Обычный синтаксис {...} позволяет создать только один объект. Конструкторы нужны для создания множества однотипных объектов.

## Функция-конструктор

Это обычная функция, но есть два соглашения

1. Начинается с большой буквы
2. Должна вызываться при помощи оператора new

### Что происходит, когда вызывается new

1. Создается пустой объект, и он присваивается this
2. Выполняется код функции
3. Возвращается значение this

## Анонимный конструктор

Можно вызывать конструктор единожды, если он не нужен в других местах:

const user = new function() {

this.name = 'Вася'

this.isAdmin = false

}

## Проверка на вызов в режиме конструктора new.target

function User() {

alert(new.target);

}

// без "new":

User(); // undefined

// с "new":

new User(); // function User { ... }

## Возврат значения из конструктора return

Вообще, конструктор всегда неявно возвращает this. Но это поведение можно перебить и вернуть свой return.

Но есть нюанс, если возвращается не объект, а примитив - то return будет игнорироваться.

## Общее

Date, Set и пр. являются функциями-конструкторами! Только встроенными

# Опциональная цепочка

Опциональная цепочка ?. останавливает вычисление и возвращает undefined, если часть перед ?. имеет значение undefined или null.

## Другие варианты применения

* С вычисляемыми свойствами объекта ?.[]
* С методами объекта ?.()

По сути - это просто безопасный способ доступа к свойствам вложенных объектов. Т.е он вернет и приостановит вычисление если левая - это undefined

# Тип данных Symbol

**Символ** - это примитивный тип данных, используется для создания уникальных идентификаторов.

Пример создания:

// Создаём символ id с описанием (именем) "id"

const id = Symbol("id");

// Аргумент - это описание. Он не обязателен, в основном используется для отладки кода

// Данный аргумент ни на что не влияет

Важно:

* Символы не преобразуются автоматически в строки
* Символы не перебираются в циклах for ... in

## Глобальные символы

* Symbol.for(key) - записывает, либо считывает по имени (ключу) символ из глобального реестра символов.
* Symbol.keyFor(sym) - получает имя символа из глобального реестра.

## Варианты использования

* Скрытые свойства объектов
  + Технически, они, конечно, не полностью скрыты. Можно получить через Object.getOwnPropertySymbols(obj) или Reflect.own(obj)
* Системные символы - для переопределения поведения, типа Symbol.toPrimitive, Symbol.iterator и т.д

# Преобразование объекта к примитиву

Если мы складываем объекты между собой, либо выполняем какие-либо математические операции - они преобразуются к примитивам.

1. Все объекты в логическом контексте являются true
2. Численные преобразования происходят когда:
   1. Вычитаем объекты
   2. Выполняем какие-либо математические операции
3. Строковые преобразования происходят когда:
   1. Мы выводим объект alert(obj)
   2. Используем объект в качестве имени свойства
   3. В других случаях, где объект используется как строка

## Варианты преобразований ("хинты")

* "string"
  + Преобразование объекта к строке
  + Используется, когда операция ожидает получить строку
    - Например alert(obj) или доступ к свойству объекта anotherObj[obj] = 123
* "number"
  + Преобразование объекта к числу
  + Используется в случае математических операций.
    - Явное преобразование Number(obj)
    - Унарный плюс +obj
    - Вычитание date1 - date2
    - Сравнения больше/меньше
* "default"
  + Происходит редко, когда оператор "не уверен", какой тип ожидать. Например, при бинарном +
  + На практике, все встроенные объекты, исключая Date, реализуют "default" преобразования тем же способом, как и "number"

## Процесс преобразования

1. Вызывает obj[Symbol.toPrimitive](hint), если метод существует и передает ему хинт.
2. Иначе, если хинт равен "string"
   1. Пробует вызвать obj.toString(), а если его нет, то obj.valueOf()
3. Если хинт равен "number" или "default"
   1. Пытается вызвать obj.valueOf(), а если его нет, то obj.toString()

## Symbol.toPrimitive

obj[Symbol.toPrimitive] = function(hint) {

// должен вернуть примитивное значение

// hint равно чему-то одному из: "string", "number" или "default"

};

Реальный пример:

let user = {

name: "John",

money: 1000,

[Symbol.toPrimitive](hint) {

alert(`hint: ${hint}`);

return hint == "string" ? `{name: "${this.name}"}` : this.money;

}

};

// демонстрация результатов преобразований:

alert(user); // hint: string -> {name: "John"}

alert(+user); // hint: number -> 1000

alert(user + 500); // hint: default -> 1500

## Методы toString/valueOf

Примера достаточно:

let user = {

name: "John",

money: 1000,

// для хинта равного "string"

toString() {

return `{name: "${this.name}"}`;

},

// для хинта равного "number" или "default"

valueOf() {

return this.money;

}

};

alert(user); // toString -> {name: "John"}

alert(+user); // valueOf -> 1000

alert(user + 500); // valueOf -> 1500