**Basic JS**

**Типи данних:**

<https://learn.javascript.ru/types-intro>

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

Методи визначення типу данних:

1. **typeof** x – повертає тип елемента

**Типи змінних:**

<https://learn.javascript.ru/variables>

<https://learn.javascript.ru/let-const>

<https://docs.google.com/presentation/d/1Qmya6GXiaic7hXzbl4M4x62cheuI5CQ_jWQTekr_5fU/edit?usp=sharing>

shortcut:

**Область видимості** визначає де в коді програми будуть доступні змінні і функції. Є 2 типи області видимості глобальна і локальна (**global** and **functiona scope**)

**Підняття (hosting) –** механізм по якому змінні і об’явлені методи рухаються вверх своєї області видимості, внезалежності від того в глобальній чи локальній вона області видимості, і об’явлення змінних / методів відбудеться перед тим як код буде виконаний. Виключити підняття в змінних буде можливо через строгий режим (‘use strict ’). JS спершу об’являє, а потім ініціалізує змінні.

Класи піднімаються, але не ініціалізуються одразу до визначення! Тобто класи хостінг ефектом не володіють.

(<https://medium.com/@stasonmars/%D1%80%D0%B0%D0%B7%D0%B1%D0%B8%D1%80%D0%B0%D0%B5%D0%BC%D1%81%D1%8F-%D1%81-%D0%BF%D0%BE%D0%B4%D0%BD%D1%8F%D1%82%D0%B8%D0%B5%D0%BC-hoisting-%D0%B2-javascript-7d2d27bc51f1> ).

**Пріорітетність змінних з однаковими іменами** – пріорітетність: змінна -> метод - серед методу і змінної із однаковими іменами – вибереться метод, як більш пріорітетний.

1. const – незмінна змінна, об’являється лише раз і не може бути переопреділеною
2. let - оптимізована версія var
   1. Має блочну область видимості (не хоститься)
   2. Видасть помилку при виклику такої змінної до об’явлення
   3. В циклі для кожної ітерації створюється нова змінна
   4. Не можна створити ще одну таку ж змінну
3. var – початкова змінна в js, яка містить певні недоліки
   1. Можемо викликати змінну до її об’явлення (underfined), змінні даного типу хостятся/піднімаютья до своєї ініціалізації (hosting)
   2. Виходить за область видимості в блоці (цикл, перевірка), але притримується функціональної області видимості
   3. В циклі перезаписуємо поточну змінну
   4. Можемо створити 2 змінні із одинаковими іменами

**Перетворення даних**

<https://medium.com/@sergeybulavyk/%D0%BF%D1%80%D0%B5%D0%BE%D0%B1%D1%80%D0%B0%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D1%82%D0%B8%D0%BF%D0%BE%D0%B2-%D0%B2-javascript-35a15ddfc333>

<https://docs.google.com/presentation/d/1aP8IjhvNjghTdlpfs1_vK7YEAMhibif84nlAlUHpzMg/edit?usp=sharing>

**Методи явного перетворення даних:**

**String(**x**) -** всі примітиви приводяться до строки

String(123) // '123'  
String(-12.3) // '-12.3'  
String(null) // 'null'  
String(undefined) // 'undefined'  
String(true) // 'true'  
String(false) // 'false'

**Boolean(**x**) –** примітиви приводяться до Boolean

Boolean('') // false  
Boolean(0) // false   
Boolean(-0) // false  
Boolean(NaN) // false  
Boolean(null) // false  
Boolean(undefined) // false  
Boolean(false) // false

Boolean({}) // true  
Boolean([]) // true  
Boolean(Symbol()) // true  
!!Symbol() // true  
Boolean(function() {}) // true

**Number(**x**) –** приведення до числа

Number(null) // 0  
Number(undefined) // NaN  
Number(true) // 1  
Number(false) // 0  
Number(" 12 ") // 12  
Number("-12.34") // -12.34  
Number("\n") // 0  
Number(" 12s ") // NaN  
Number(123) // 123

Повертає NaN, якщо обрізана строка не придставляє собою число. Якщо строка буде пустою, то отримаємо 0.

При застосуванні до null чи undefined, чисельного приведення даних не відбудеться, так як null може рівнятись тільки null или undefined, і нічому іншому.

null == 0 // false, null is not converted to 0  
null == null // true  
undefined == undefined // true  
null == undefined // true

**Методи неявного приведення даних:**

**Строкове:**

* Неявне перетворення буде створене лише через ‘+’ і наявність пустої строки при вичислені, інакше це буде перетворення до числа
* Не явне перетворення в строку: **‘’ + value**

**Булеве:**

Неявне перетворення відбувається в логічному контексті if (val) { … } чи при застосуванні логічних операторів (|| && !).

На замітку, логічні оператори, такі як || и && видають булеве перетворення під капотом, але при цьому [завжди](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators#Logical_operators) повертають оригінальне значення операнідів, навіть якшо вони не являються белевеми

**Числове:**

* Оператори порівняння (>, <, <=,>=)
* Бінарні оператори (| & ^ ~)
* Арифметичні оператори (- + \* / % ). Зверніть увагу, що бінарний оператор + не не викликає чисельного перетворення, якщо один з операндів являється строкою
* Унарний оператор +
* Оператор нестрогої рівності == (включаючи !=). Зверніть увагу, що даний оператор не викликає чисельного перетворення, якщо обидва операнди являються строками

В новому синтаксисі з’явилась можливість перевіряти із присвоюванням:

&&=, ||=, ??=

a &&= b (a = b)

**Тернарні оператори:**

1. <https://learn.javascript.ru/comparison>
2. <https://learn.javascript.ru/ifelse>
3. <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Conditional_Operator>
4. <https://docs.google.com/presentation/d/1jF-QOejhsn0TVsV1wCuxH7fqok0jll-Mb3OSiMhePBQ/edit?usp=sharing>
5. <https://docs.google.com/presentation/d/1nRs0i7rRydj1K3HXO-64B3gK9ZpeMjafpLX0g0riOBM/edit?usp=sharing>

**Цикли:**

* 1. <https://docs.google.com/presentation/d/19_aP7liLVCaMOpWCTDbfpBkHeerRJxPTP2b6HmFK8xU/edit?usp=sharing>
  2. <https://learn.javascript.ru/while-for>
  3. <https://learn.javascript.ru/object-for-in>
  4. <https://frontender.info/es6-in-depth-iterators-and-the-for-of-loop/>

**Числа:**

1. <https://learn.javascript.ru/number>
2. <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number>
3. <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math>
4. <https://docs.google.com/presentation/d/11i9ipXc6duAgYDn5QxxbT7yr0XwyWGO8PMNYhU8Gbhg/edit?usp=sharing>

**Строки:**

1. <https://learn.javascript.ru/string>
2. <https://developer.mozilla.org/uk/docs/Web/JavaScript/Reference/Global_Objects/String>

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

1. <https://docs.google.com/presentation/d/12SbhHaPH9pueWmnbapi0yiyWp4gUH6f9Nr5oQa62Hsk/edit?usp=sharing>
2. <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals>

**Об’єкти:**

1. <https://learn.javascript.ru/object>
2. <https://developer.mozilla.org/ru/docs/Learn/JavaScript/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D1%8B/%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D1%8B>
3. <https://docs.google.com/presentation/d/1xokG57sfQBPh7tIL9_utYwbmkzOV7fHe5Gmgg1xidzE/edit?usp=sharing>

**Динамічні ключі** – ключі задані через змінну і записані в [] ->

const nameField = ‘city’

obj = {**[nameField]**: 12}

**Масивив:**

1. <https://learn.javascript.ru/array>

**Функції в JS**

1. <https://learn.javascript.ru/function-basics>
2. <https://docs.google.com/presentation/d/1dHzcwDWIv_ybcmcI60UaalmB2o1brJOXhthDbJ4wecw/edit#slide=id.g5dbb21c589_2_58>
3. <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/Default_parameters>

Особливості методів js:

1. Зберігаються в змінній, об’єкті, масиві
2. Передаються, як аргументи в іншу функцію
3. Можуть повертатись із функцій

**CallBack functions**

1. <https://medium.com/nuances-of-programming/%D0%BF%D0%BE%D0%B4%D1%80%D0%BE%D0%B1%D0%BD%D0%B5%D0%B5-%D0%BE-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F%D1%85-%D0%B2%D1%8B%D1%81%D1%88%D0%B5%D0%B3%D0%BE-%D0%BF%D0%BE%D1%80%D1%8F%D0%B4%D0%BA%D0%B0-2d94ace3e96c>
2. <https://medium.com/nuances-of-programming/%D0%B8%D0%B7%D1%83%D1%87%D0%B0%D0%B5%D0%BC-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8-%D0%B2%D1%8B%D1%81%D1%88%D0%B5%D0%B3%D0%BE-%D0%BF%D0%BE%D1%80%D1%8F%D0%B4%D0%BA%D0%B0-%D0%B2-javascript-c23daf53a5c0>

**CallBack** - функція, яка передається в функцію в якості аргумента, **Callback** – функція, яка повинна бути виконана після того, як інша функція завершила виконання. **Callback** – дає нам гарантію, що деякий код не виконається до моменту, поки другий код не завершить виконання.

**Декораторы** — це фнукції вищого порядку, які приймають функцію у вигляді аргумента і повертають іншу функцію. Функція, яка повертається – це модифікована функція з аргументами. Але фішка в тому, що ми не міняємо код оригінальної функції. **Функції декоратори** потужна штука для створення довільної поведінки вже існуючих функцій без зміни коду поточних.

**Лямбда функції** — це функції, які використовуються, як значення.

**Функціональне програмування** – спосіб програмування при якому ви можете передавати функції в якості параметрів іншим функціям. За таким програмуванням ми пишемо код з точки зору функцій

**Часткове застосування** – це передача в функцію меншої кількості аргументів, ніж вона очікує.

Якщо функція повертає функцію, то ми можемо одразу викликати внутрішню функцію синтаксисом: **function func(){ return func2() {}} -> func()()**;

**Функціональне програмування** – тип написання коду при якому ми передаємо функції в якості параметра іншій функції

**Контекст виклику this:**

* 1. <https://docs.google.com/presentation/d/1-qAPOQqQS73az6uk5t5mwrrCin45LDAVmgPTIYRYybs/edit#slide=id.g5dbb3866c6_2_116>
  2. <https://habr.com/ru/company/ruvds/blog/419371/>
  3. <http://ryanmorr.com/understanding-scope-and-context-in-javascript/>
  4. <https://learn.javascript.ru/call-apply-decorators>
  5. <https://tproger.ru/translations/javascript-this-keyword/>
  6. <https://medium.com/@stasonmars/%D0%BF%D0%BE%D0%B4%D1%80%D0%BE%D0%B1%D0%BD%D0%BE-%D0%BE-%D1%82%D0%BE%D0%BC-%D0%BA%D0%B0%D0%BA-%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%B0%D0%B5%D1%82-this-%D0%B2-javascript-a13b4b6ec9ac>
  7. https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/this

Оточення в якому виконується функція називається – **контекстом виклику функції** (функція викликана в контексті деякого оточення: об’єкт чи браузер)

**This** – об’єкт контексту, завдяки якому в середині будь-якої функції можна получити доступ до об’єкту, який викликав функцію. Контекст завжди є значенням ключового слова this, яке посилається на об'єкт, що «володіє» кодом, виконуваним в поточний момент. В строгому режимі (**‘ use strict ’**) – при виклику функції в глобальному оточені this матиме значення undefined, а не window

**Chaning** – прийом при якому методи об’єкта можна викликати через точку один за одним, будуючи ланцюг викликів. Для реалізації даного функціоналу метод об’єкта завжди повинен повертати цей об’єкт (return this).

**Втрата контексту** – коли функція викликається в контексті відмінному від очікуваного.

**В стрілочних фукнціях this** - Не важно як стрілочна функція буде викликана, її значення this завжди буде вказувати на глобальну область видимості, навіть якщо вона буде викликана в якості метода об’єкту, чи з використанням call, apply, bind. Стрілочна функція не створює свій контекст виконання, а получає this із зовнішньої функції, в якій вона була визначина. This стрілочної функції неможливо переопреділити. Оскільки значення this в стрілочних функціях оприділяються лексичним середовищем – правила строгого режиму (‘ use strict ’) на дане this не діють

**В методах подій – this** буде посилатись на елемент з якого починається подія

**При передачі метода об’єкта змінній** і ініціалізації його this – буде window

**This при виклику конструктора** – належить щойно створеному об’єкту. Використання конструктора чи класу без new – задасть this значення window

**Коли this використовується всередині об'єкта**, це ключове слово посилається на сам об'єкт.

**В вложених об’єктах** варто пам'ятати про те, що ключове слово this ставитися до того об'єкту, в методі якого воно використовується.

--- Якщо ми хочемо використати this в контексті анонімного методу певного об’єкта, то this буде мати контекст даного метода, а не об’єкта в якому цей метод знаходиться. Щоб усе працювало коректно, потрібно:

1. для даного методу викликати bind(this) – в контексті об’єкта
2. або зберегти контекст в змінну і викоритовувати її вже в даному методі, прив’язуючись до об’єкта, а не до методу.
3. Або використовувати стрілочні методи, які не створюють власного контекстуі

this – завжди посилається на об’єкт, який знаходиться зліва від точки, якщо він об’явлений в об’єкті чи методі (не стрілочній фукнції)

**obj**.name

**Методи переопреділення контексту виклику call(), apply(), bind() :**

1. <https://medium.com/@stasonmars/%D0%BF%D0%BE%D0%B4%D1%80%D0%BE%D0%B1%D0%BD%D0%BE-%D0%BE-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B0%D1%85-apply-call-%D0%B8-bind-%D0%BD%D0%B5%D0%BE%D0%B1%D1%85%D0%BE%D0%B4%D0%B8%D0%BC%D1%8B%D1%85-%D0%BA%D0%B0%D0%B6%D0%B4%D0%BE%D0%BC%D1%83-javascript-%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D1%87%D0%B8%D0%BA%D1%83-ddd5f9b06290>
2. <https://learn.javascript.ru/call-apply>
3. <https://learn.javascript.ru/bind#%D0%BA%D0%B0%D1%80%D1%80%D0%B8%D0%BD%D0%B3>
4. <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/this>
5. <https://learn.javascript.ru/call-apply-decorators>

Різниця між call/apply i bind – в тому, що перші два викликаються контекст для необхідного об’єкта, а bind не викликає метод, а повертає його з уже строго прив’язаним контекстом

**Часткове застосування функції –** ми викликаємо функцію із меншою кількістю параментрів, ніж вона очікує, вона ж в свою чергу повертає функцію, яка приймає решту аргументів.

**Карірування** – використання функції, яка повертає функцію з уже частково заданими параметрами. Ця функція має доступ до збережених агументів і змінних зовнішньої функції.

(<https://medium.com/devschacht/functional-reactive-ninja-partial-application-of-functions-99fc21d629ff> )

Одним із способів реалізувати **часткове застосування функції** є використання методу bind().

function mul(a, b) {

return a \* b;

}

let double = mul.bind(null, 2);

alert( double(3) ); // = mul(2, 3) = 6

alert( double(4) ); // = mul(2, 4) = 8

alert( double(5) ); // = mul(2, 5) = 10

Мінуси даного методу:

1. Function.bind() – завжди повертає іншу функцію, навіть, якщо ми передамо усі параметри базової, тому ми не знаємо, коли зупинитись
2. Перший параметр bind() – для задавання контексту виконання обов’язковий і нам доводиться ставити заглушку null, приєднуючи контекст

Але якщо ми хочемо зафіксувати деякі аргументи, але не контекст this – краще використовувати **карірування**.

let add = x => y => x+y;

let increment = add(1);  
let incrementBy2 = add(2);

console.log('Increment 3 by 1:',increment(3));  
//=> Збільшуємо 3 на 1: 4  
console.log('Increment 3 by 2:',incrementBy2(3));  
//=> Увеличиваем 3 на 2: 5

Метод обгортка слугує для задання базових параметрів, а метод, який повертається приймає змінний параметр.

Переваги даного методу:

1. Карірування завжди зроблено так, що вона завжди повертає нішу функцію, яка приймає лише 1 параметр
2. Функція, яка карірується завжди запам’ятовує передаваємі аргументи через замикання
3. Карірувана функція завжди генерує одну і туж функцію для одних і тих самих вхідних данних.

Способи функціонального програмування:

1. **Композиція** – це об’єднання двох і більше функцій для створення нової функції.
2. **Функція в безточковому стилі (комбінаторне програмування)** – парадигма програмування, не протребуюча явного згадування аргументів, які оприділяються функцією і викоритовуються замість змінних комбінаторів і композицій.

**Каррірування** готує функції, щоб приймати тільки дані в якості аргументу (інші аргументи частково застосовуються заздалегідь), а **композиція** допомагає в об'єднанні цих частково застосованих функцій, щоб дані могли проходити через них.

**Стрілочні функції:**

1. <https://docs.google.com/presentation/d/1FZLdHz3NqeKc-h_YsIT4IYbflbz__HQ0-L9knu7lahY/edit#slide=id.g5dbbfdbf65_2_126>
2. <https://medium.com/@KucherDev/%D0%BA%D0%BE%D0%B3%D0%B4%D0%B0-%D0%B8-%D0%BF%D0%BE%D1%87%D0%B5%D0%BC%D1%83-%D1%81%D1%82%D0%BE%D0%B8%D1%82-%D0%B8%D1%81%D0%BF%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D1%8C-%D1%81%D1%82%D1%80%D0%B5%D0%BB%D0%BE%D1%87%D0%BD%D1%8B%D0%B5-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8-es6-3135a973490b>
3. <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/Arrow_functions>

Стрілочні фукнції не мають свого об’єкт arguments, тому в тілі стрілочних функцій arguments буде посилатися на змінну в оточуючій області, тому для **створення стрілочного методу з динамічною кількістю параметрів** можемо використовувати rest параметри

function foo() {

var f = (...args) => args[0];

return f(2);

}

Для використання операторів порівняння краще використовувати даний синтаксис

callback = callback || () => {};

// SyntaxError: invalid arrow-function arguments

callback = callback || (() => {}); // ok

Коли не варто використовувати стрілочні функції:

1. Як методи об’єкта з використанням this
2. Якщо потрібно, щоб контекст був динамічним (оброботчик подій із викоританням this)
3. Коли код стає менш четаємим

Коли варто використовувати стрілочні функції:

1. Коли від this потребується прив’язка до контексту, а не до самої функції.

**Замикання:**

1. <http://dmitrysoshnikov.com/ecmascript/ru-javascript-the-core/#stek-kontekstov-ispolneniya>
2. <http://dmitrysoshnikov.com/ecmascript/ru-chapter-4-scope-chain/>
3. <https://habr.com/ru/company/hexlet/blog/266443/>
4. <https://learn.javascript.ru/closure>
5. <https://developer.mozilla.org/ru/docs/Web/JavaScript/Closures>
6. <https://docs.google.com/presentation/d/17XkkZaPwrNgApxz_07RymNPBYFTMK4lmZLjKtkHUEu8/edit#slide=id.g5dad253389_2_120>
7. <https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-closure-b2f0d2152b36> - english
8. <https://medium.com/dailyjs/i-never-understood-javascript-closures-9663703368e8> - english
9. <https://medium.com/@sshambir/%D0%BE%D1%81%D0%B2%D0%B0%D0%B8%D0%B2%D0%B0%D0%B5%D0%BC-%D0%B7%D0%B0%D0%BC%D1%8B%D0%BA%D0%B0%D0%BD%D0%B8%D1%8F-%D0%B2-javascript-5b83267ef7d1>

**Замикання** – це функція яка посилається на незалежні змінні. Іншими словами функція визначена в замиканні ‘запам’ятовує’ оточення в якому була створена. Якщо звичайна функція завершує свою роботу, то збірник мусору забираю scope-об’єкт на який ніхто не зсилається. Але якщо ми визначимо вложену функцію і вернути її, то функція-об’єкт (локальна область видимості) завжди посилається на scope-об’єкт в якому вона була створена і навіть при завершенні зовнішньої функції її scope-об’єкт не буде очищений збірником мусору, навіть коли дана функція поверне значення, так як на цей об’єкт все ще є посилання. Таким чином коли ми задаєм вложену функцію, вона получає посилання на поточну область видимості зовнішньої функції, тим самим зберігає її від очищення.

Один із об’єктів ланцюга областей видимості містить то лексичне оточення (батьківський об’єкт змінних) в якому функція зароджується, змінні із вищестоящих контекстів , при подальшій активації функції будуть шукатися саме в цьому лексичному ланцюгу об’єктів змінних.

**Лексичне середовище** – внутрішній (прихований) об’єкт пов’заний з кожною виконуваною функцією чи блоком коду, який складається із двох частин. Перша Enviroment Record – об’єкт, в якому, як властивість зберігаються всі локальні змінні, а також this. Будь-яка змінна це властивість Enviroment Record і змінюючи змінну ми міняємо властивість даного об’єкта. Друга – посилання на зовнішнє лексичне оточення, те яке відповідає коду ззовні.

В процесі виклику функції в нас є два лексичних оточення внутрішнє, для викликаної функції (функціональний об’єкт) і зовнішнє (глобальне). У внутрішнього лексичного середовища є посилання на зовнішнє. Коли код хоче получити доступ до змінної – спершу відбувається пошук по внутрішньому лексичному оточенні, після в зовнішньому і так по ланцюгу scope-об’єктів догори. Функція получає поточне значення зовнішніх змінних, тобто їх останнє значення.

Нове лексичне середовище створюється кожен раз, коли функція виконується. І якщо фукнція викликаєтся декілька раз, то для кожного виклику буде створено нове лексичне середовище з своїми специфічними для цього виклику локальними змінними і параметрами

**Вільна змінна** – змінна, яка не знаходиться в локальній області функції, яка її використовує (ні як змінна, ні як параметр). Якщо функція не знаходить змінну в локальній області видимості, то шукає її в батьківських областях видимості. Для пошуку вільних змінних використовується ланцюг області видимості.

**Контекст виконання** – абстрактна концепція, в рамках якої специфікація js пояснює модель виконання коду після запуску.

**Стек** – структура даних, діюча по принципу ‘перший зайшов – вийшов останнім’.

**Диструктуризація:**

1. <https://docs.google.com/presentation/d/1VIpoedDIK_c-vcZilmIfhjgNIufHkTmIYiftx-IEY-4/edit#slide=id.g5db9d316cc_2_112>
2. <https://medium.com/@stasonmars/%D0%B4%D0%B5%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F-%D0%B2-es6-%D0%BF%D0%BE%D0%BB%D0%BD%D0%BE%D0%B5-%D1%80%D1%83%D0%BA%D0%BE%D0%B2%D0%BE%D0%B4%D1%81%D1%82%D0%B2%D0%BE-b865bb71f376>
3. <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment>
4. <https://frontender.info/es6-in-depth-destructuring/>

**Диструктуруючі присвоєння** – спосіб видобутку даних з масива чи об’єкта спеціальним синтаксисом. Спосіб розбити складні структури на простіші елементи. Для видобутку даних із масива використовують [], для об’єкта {}.

**const** [ name1, name2 ] = [ 'John', 'Mike', 'Abraham', 'Piter' ];  
console.log(name1, name2); //  John Mike

**Для пропуску непотрібних елементів** потрібно в диструктуруючому присвоєнні пропустити дані елементи:

**const** [ , name2, name3 ] = [ 'John', 'Mike', 'Abraham', 'Piter' ];  
console.log(name2, name3);  // Mike Abraham

**Для отримання решти елементів** після обирання потрібних – можна **використати rest оператор** ‘**…name’.** В диструктурованому присвоєнні rest оператор необхідно використовувати останнім.

**const** [ name1, name2, **...other** ] = [ 'John', 'Mike', 'Abraham', 'Piter' ];  
console.log(name1, name2); // John Mike  
console.log(other); //["Abraham", "Piter"]

Якщо в диструкторованому присвоєнні більше параментрів, ніж є елементів в даних об’єктах, то параметрів, яким не вистачило елемента отримують undefined. Для уникнення даної ситуації можна задати параметр по замовчуванню.

**При диструктуризації об’єкта** в фігуних дужках вписуємо назви полів, які хочемо отримати з об’єкта. **Для переопреділення імен** полів можна використовувати перейменування через ‘:’.

**const** figure = { width: 10, height: 12, type: 'square' };  
**const** { width: w, height: h, type: t } = figure;

console.log(w, h, t); // 10 12 "square"

**Диструктуризація** доступна і **для вложених об’єктів/масивів.** Для диструктуризації вложеного об’єкта використовується параметр перейменування ‘:’ і скобки в залежності від типу об’єкта ({}, []). Диструктуризацію можна провести будь-якої сладності і глибини.

При передечі диструктуризованого об’єкта параметром функції, для уникнення помилок, мотрібно вказати параметр по замовчуванню

function *getBaseInfo*( { name = 'John', age = 18} = **{}** ) {

Серед додаткового функціоналу диструтуризації можна:

1. Обмін простими значеннями

let width = 300;

let height = 400;

const landscape = true;

if (landscape) {

**// Меняем значения переменных**

[width, height] = [height, width];

**Проблеми диструктуризацї методів з контекстом this**

Коли ми диструктуризуємо метод об’єкта, що містить в собі this, то прив’язка до контексту об’єкта, з якого робиться диструктуризація втрачається і нам потрібно даний метод викликати в контексті даного об’єкта, щоб повернути контекст

**const {count()} = obj**

**count().call(obj)**

Якщо ми міняємо назву методу то контекст викликаємо вже в нової назви методу

**const {count: addressCount} = obj**

**addressCount().call(obj)**

**Дескріптори, гетери, сетери**

1. <https://learn.javascript.ru/descriptors-getters-setters>
2. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty?retiredLocale=uk>

# **Об’єкт Обгортки**

1. <https://javascriptrefined.io/the-wrapper-object-400311b29151>

**Всякий раз, коли ви намагаєтеся отримати доступ до властивості рядка str, JavaScript призводить значення рядка до об'єкту за допомогою нового рядка (str). Цей об'єкт називається об'єктом-оболонкою.**

**JavaScript при необхідності приводить об'єкти-обгортки в загорнуті значення примітивів. == рівність розглядатиме значення і його об'єкт-оболонку як рівні, в той час як оператор суворого рівності === розглядатиме їх як різні об'єкти.**

**var a = 'привіт'; // примітив**

**var b = new String ( 'hello'); // об'єкт-обгортка**

**typeof a; // "рядок"**

**typeof b; // "об'єкт"**

**a == b // true**

**a === b // false**