**JavaScript** – є синхронною і однопотоковою мовою програмування високого рівня, реалізованою згідно стандарту ECMA Script.

**Стандарт ECMA Script (ES)** – це специфікація, яка визначає синтаксис, типи даних, оператори, об'єкти та інші елементи JavaScript.

**Мова JavaScript використовується для:**

* розробки клієнтської частини веб-додатків, що працюють у браузері.
* створення односторінкових та прогресивних вебзастосунків (React, AngularJS, Vue.js)
* програмування на боці сервера (Node.js(Express.js ))
* стаціонарних застосунків (Electron, NW.js)
* мобільних застосунків (React Native, Cordova)
* сценаріїв в прикладних програмах (наприклад, в програмах зі складу Adobe Creative Suite чи Apache JMeter )
* всередині PDF-документів тощо.

**Основні особливості JavaScript:**

* Динамічна типізація: Змінні в JavaScript не мають статичного типу даних і можуть змінювати свій тип під час виконання.
* Об'єктно-орієнтоване програмування: JavaScript підтримує об'єктно-орієнтовану парадигму, що дозволяє створювати та використовувати об'єкти, класи, успадкування та поліморфізм.
* Функціональне програмування: JavaScript також підтримує функціональне програмування, дозволяючи використовувати функції як значення, передавати їх як аргументи, повертати функції з інших функцій тощо.
* Маніпуляція DOM: JavaScript надає доступ до об'єктної моделі документа (DOM), що дозволяє змінювати структуру та вміст HTML-елементів на веб-сторінці.
* Асинхронне програмування: JavaScript має механізми асинхронного програмування, такі як колбеки, проміси та асинхронні функції, для ефективної обробки операцій.

**Основні складові частини виконання програми JavaScript**

Середовище у якому браузер виконує програми JavaScript - **Execution context** (контекст виконання).

Компоненти контексту виконання:

1. Компонент пам’яті - **Memory** (колекція пар ключ-значення : назва\_змінної : значення\_змінної).
2. Компонент коду - **Thread of Execution** (місце де виконуються команди).

|  |  |
| --- | --- |
| **Memory** | **Thread of Execution** |
| https://lh3.googleusercontent.com/WKNDus_TkwsP9G9FyJBVSG8PNV2ZVDXcyMMOczs3Kg6NNfFcS1ch8QgbyORxctz1BKVpsX09oBhws1JH-ybiDBNJuLIVp6GyLACJf5ENzssrYFMuwTPEfxDVsVVaNMCEFiOuIZdoPoz0b5eq7plc_w=s2048 | https://lh4.googleusercontent.com/eoxLV1YgMSoDn9whMnvCk4jLW_IfSdBrArKB6ltw0IIzMAPAWOW4937bzFx_T4X4sE0uXXou2br-gHfJ64nkbBzW_pTT6M2Ajpv4EhvkaVKfb3dEPwHmsOAsmsViGq_ixyMTLiwiT1BAKVCGaY-W6Q=s2048 |

Фази виконання програм JavaScript

1. Фаза **виділення пам’яті** (виділення пам’яті для змінних і функцій).
2. Фаза **виконання команд** (поступове виконання команд по одній за раз).

|  |  |
| --- | --- |
| **Фаза виконання** | **Стан контексту виконання** |
| 1) **Виділення пам’яті** (змінні мають початкове значення - **undefined**) | |  |  | | --- | --- | | **Memory** | **Thread of Execution** | | https://lh4.googleusercontent.com/JXWDZ-35wAk6lkbEQyYxacyypZQwUdu4RddU6WkqM6M70ecHl54j9vDR4K3TvqASGzZpn4q8_5SXQNAjjQ5Gy9vSjDrYtV7gtUHm2QmUXw98iH98b_-mGW-L8amFHY5FlMX3EeiW-NoeG-NoeohAvw=s2048 |  | |
| 2) **Виконання команд** | |  |  | | --- | --- | | **Memory** | **Thread of Execution** | | https://lh3.googleusercontent.com/nwHMoNS6d7TVZyG2g7jXxxha8PnyqqK7Zo4ryoO6l9bawpEI4mdDT8qNi5lKJ6IauFMtuzuxnGzrvY15wPzBPWrWhc4mLXpzaz1vPlXD4bNMcErqg6bCOm81TBKyRXlCZlmPMwYbI3Qd8vDisX0S4A=s2048 | https://lh5.googleusercontent.com/5_ag_Ct0rryZDdgp3HmWWsnMQWrATDvi78YQrV_irCxbmZUSCk_gPxZTF2azGkuOH3at_pT9v_UeI30gUnzNC2LwB5SusxCby2lEoroBF8dF4oO15gU2yArO3Aq9mi5FgXBeMO0UMpYJZfoqeyR8Ag=s2048 | |

Підключення та порядок виконання скриптів

1. **Блокуючий обробку сторінки.**

|  |  |
| --- | --- |
| <script src=”script1.js”>  <script src=”script2.js”> | Сторінка **html** заванатжується. Як тільки черга доходить до  **<**script src=**”script1.js”>**, завантаження **html** призупиняється, браузер починає завантажувати і відразу здійснювати файл **”script1.js”,** потім завантажує і відразу здійснює **”script2.js”**, і тільки після цього браузер продовжить дозавантажувати **html**. |

1. **Неблокуючий обробку сторінки.**

|  |  |  |
| --- | --- | --- |
| <script **async** src=”script1.js”>  <script **async** src=”script2.js”> | Сторінка **html** заванатжується. Як тільки черга доходить до  **<**script **async** src=”script1.js”**>** **він асинхронно завантажується**, разом з завантаженням HTML-сторінки. **Скрипт почне виконуватись відразу**, незалежно від стану обробки сторінки або інших скриптів.  Якщо скриптів декілька тоді **скрипти виконуються у будь якому порядку** (хто встиг швидше завантажитись), незалежно від їх розташування у структурі html. | |
| <script **defer** src=”script1.js”>  <script **defer** src=”script2.js”> | | Сторінка **html** заванатжується. Як тільки черга доходить до  **<**script **defer** src=”script1.js”**>** **він асинхронно завантажується**, разом з завантаженням HTML-сторінки. **Скрипт почне виконуватись тільки після завершення завантаження HTML-сторінки**.  **Скрипти будуть виконуватись почерзі**, згідно їх розташування у структурі html. |
| <script **defer** src=”script1.js”>  <script **async** src=”script2.js”> | | Спочатку завантажиться і виконається <script **async** src=”script2.js”>  Потім завантажиться <script **defer** src=”script2.js”>  Потім завантажиться HTML-сторінка  Потім виконається <script **defer** src=”script2.js”> |

Типи даних JavaScript

* Рядки (**Strings**): Використовуються для збереження текстової інформації (символи, слова, речення). Наприклад: "Привіт, світ!".
* Числа (**Numbers**): Використовуються для збереження числових значень як цілик так і дробових. Наприклад: 42, 3.14.
* Булеві значення (**Boolean**): Мають лише два можливих значення: true (істина) і false (хиба). Використовуються для логічних операцій та умовних перевірок.
* Об'єкти (**Objects**): Використовуються для збереження складних структур даних, які містять властивості і методи. Об'єкти створюються за допомогою фігурних дужок {}.

Наприклад: {name: "John", age: 30}.

* Масиви (**Arrays**): Використовуються для збереження колекцій значень впорядкованого списку. Масиви створюються за допомогою квадратних дужок []. Наприклад: [1, 2, 3, 4].
* **Null**: Використовується для вказання відсутності значення. Він є окремим типом даних і має єдине значення - null. Присвоюється змінній, коли потрібно показати, що вона не має значення або потребує очищення.
* **Undefined**: один з примітивних типів даних, який вказує на відсутність присвоєного значення змінній або властивості об'єкта.
* **Symbol**: Це новий тип даних, введений у стандарті ECMAScript 6. Використовується для створення унікальних ідентифікаторів. Начало формы

Визначення типу даних **typeof**

**Оператор "typeof"** використовується для визначення типу даних змінної, значення або виразу в JavaScript. Він дозволяє дізнатись, до якого саме типу даних належить певний об'єкт чи значення.

Оператор "typeof" повертає “рядок”, який відповідає типу даних. Наприклад, він може повернути "**number**" для числа, "**string**" для рядка, "**boolean**" для булевого значення, "**object**" для об'єкта, "**function**" для функції та так далі.

let x = 42

console.log(typeof x) // Виведе "number"

let message = "Привіт, світ!"

console.log(typeof message) // Виведе "string"

let a1

console.log(typeof a1) // Виведе "Underfined"

let isActive = true

console.log(typeof isActive) // Виведе "Boolean”

**Літерали**

**Літерал** - це явно вказане значеннчя якогось типу. Літерали використовуються для створення конкретних значень для різних типів даних, таких як числа, рядки, булеві значення, масиви, об'єкти та інші.

* Літерал числа ( **цілого** або **дійсного** типу ): 42, -10, 3**.**14
* Літерал рядка ( символьний ): "Привіт, світ!", **'**JavaScript**'**, **`**Рядок зі змінною **${**variable**}`** (шаблонний рядок).
* Літерал булевого значення ( логічний ): true, false.
* Літерал масиву: [1, 2, 3], ['яблуко', 'банан', 'помаранч'].
* Літерал об'єкта: { name: 'John', age: 30 }.
* Літерал регулярного виразу: /pattern/, /[A-Za-z]+/.

Спецсимволи JS ескейп полідовності

**\b**: Використовується для представлення символу зворотнього виведення (backspace).

**\f**: Використовується для представлення символу переходу на нову сторінку (form feed).

**\r**: Використовується для представлення символу повернення каретки (carriage return).

**\n**: Використовується для представлення символу нового рядка (line feed).

**\v**: Використовується для представлення символу вертикальної табуляції (vertical tab).

**\t**: Використовується для представлення символу горизонтальної табуляції (horizontal tab).

**\'**: Використовується для екранування символу одинарної лапки всередині рядка, який вже містить одинарні лапки.

**\"**: Використовується для екранування символу подвійної лапки всередині рядка, який вже містить подвійні лапки.

**\XXX**: Восьмеричний спецсимвол. Використовується для представлення символу з восьмеричним кодом XXX.

**\xXX**: Шістнадцятковий спецсимвол. Використовується для представлення символу з шістнадцятковим кодом XX.

**\uXXXX**: Юнікод-символ. Використовується для представлення символу Юнікоду з кодом XXXX, де XXXX - код символу в шістнадцятковому форматі.

**\u{XXXXX}**: Розширений юнікод-символ. Використовується для представлення символу Юнікоду з розширеним кодом XXXXX, де XXXXX - код символу в шістнадцятковому форматі.

Опис змінних

У JavaScript **змінна** - це іменоване місце для зберігання даних. Змінні використовуються для зберігання та маніпулювання значеннями певних типів даних, таких як числа, рядки, булеві значення, об'єкти тощо.

**Оголошення змінної в JavaScript:**

let myNumber // Оголошуємо (створюємо) з мінну з назвою **myVariable**. Зараз її значення = **Undefined**

myNumber = 10 // За допомогою оператора присвоєння (**=**) присвоїмо їй значення **10**

let myString = "Hello" // Можна оголошувати змінні і присвоювати їм значення одночасно

myNumber = "World" // JS є динамічно типізованою мовою - можна змінювати значення та тип змінної

Правильна назва змінної (ідентифікатор)

* Повинна бути унікальною, та містити тільки букви латинського алфавіту, цифри та символи – **$** та **\_**
* Не може починатись з цифр та співпадати з зарезервованим словом

|  |  |
| --- | --- |
| Непривильні ідентифікатри | my id // містить пробіл  Ok! // містить недозволений знак оклику **!** Ab-c // містить недозволений знак тире **-**  12th // починається з цифри  100 // починається з цифри  for // зарезервоване слово |

CamelCase

CamelCase – стиль (нотація) формування ідентифікаторів при якому декілька слів пишуться разом (без пробілів) і при цьому кожне слово починається з великої літери.  
Розрізняють два види CamelCase нотації у залежності від того, велика чи маленька перша літера:

* UpperCamelCase (РascalCase) - перша літера велика (для позначення типів даних чи класів)
* lowerCamelCase - перша літера маленька (для позначення назв змінних)

Snake case

Snake case – стиль (нотація) формування ідентифікаторів, коли окремі слова у назві відділяються символами нижнього підкреслення **\_ .** При цьому слова можуть писатися як маленькими, так і великими літерами.  
У багатьох мовах програмування з використанням цього стилю прийнято формувати ідентифікатори **костант**

* SNAKE\_CASE - всі літери великі (для позначення констант)

Область видимості змінної

Змінні в JavaScript можуть мати різні **області видимості**, такі як **глобальна** (видима в усій програмі) або **локальна** (видима лише у межах певної функції чи блоку).

**Локальна (функціональна) область видимості**: Змінні, оголошені всередині функції (включаючи параметри функції), мають функціональну область видимості. Це означає, що ці змінні будуть доступні всередині цієї функції, а також всередині будь-яких вкладених функцій (якщо такі є). Змінні з функціональною областю видимості не будуть доступні поза функцією, в якій вони були оголошені.

**Локальна (бочна) область видимості:** Змінні, оголошені всередині блока коду **{** … **}** (наприклад, умовного оператора if, циклу for) мають блочну область видимості. Це означає, що ці змінні будуть доступні лише всередині цього блока та будь-яких вкладених блоків. Вони не будуть доступні поза цим блоком. Це правило стосується змінних, оголошених з використанням let або const.

Змінні, оголошені з використанням **let**, можуть змінювати свої значення.  
Змінні, оголошені з використанням **const**, є незмінними (константами) і не можуть змінювати свої значення.

**function** exampleFunction() {

**var** firstVariable = "First" // Змінна **з локальною** областю видимості

**if** (true) {

**var** secondVariable = "Second" // Змінна **з локальною** областю видимості

**let** minLenght = " minLenght" // Змінна **з блочною** областю видимості

**const** MAX\_LENGHT = "MAX\_LENGHT" // Константа **з блочною** областю видимості

console.log(firstVariable) // Виведе "First"

console.log(secondVariable) // Виведе "Second"

console.log(minLenght) // Виведе "minLenght"

console.log(MAX\_LENGHT) // Виведе MAX\_LENGHT"

}

console.log(firstVariable) // Виведе "First functional scope "

console.log(secondVariable) // Виведе " Second functional scope "

console.log(minLenght) // Помилка: minLenght не доступна за межами блока

console.log(MAX\_LENGHT) // Помилка: MAX\_LENGHT не доступна за межами блока

}

console.log(firstVariable) // Помилка: varVariable не доступна за межами функції

console.log(secondVariable) // Помилка: varVariable не доступна за межами функції

Способи **виведення даних**

* **Використання функції console.log()**: Цей метод дозволяє виводити дані у консоль браузера або середовища виконання JavaScript.

**console.log**("Hello world”)

* **Використання window.alert()**: Ця функція виводить повідомлення в модальному вікні браузера.

**alert**("Hello world”)

Діалогове вікно приймає дані будь-яких типів, та представляює їх у string.

* **Використання** **document.write()** для виведення даних прямо на HTML сторінці.

**document.write**("Hello world”)

Способи **введення даних**

**prompt**(**message**, **default**) - вспливаюче вікно з повідомленням для введення даних. Має текстове повідомлення і місце для введення даних.

* **message** - це текстове повідомлення, яке відображається користувачу.
* **default** - це необов'язкове значення, яке встановлюється в поле введення за замовчуванням.

Коли функція prompt викликається, виконання коду блокується, доки користувач не введе дані та не натисне кнопку "OK" або "Скасувати".

let name = prompt("Введіть ваше ім'я:", " Анонім")

**Важливо |** Значення, отримане з функції prompt, завжди є **рядком**. Якщо потрібно використовувати це значення як число, можна скористатися функціями - **parseInt** або **parseFloat**.

**Перетворення** рядкана число

**parseInt**(string, radix) **–** функція використовується для перетворення рядка (string) на **ціле число** (**integer**).

* **string -** рядок, який потрібно перетворити на ціле число.
* **radix**  - вказує у якій системі числення заданий рядок (необов'язковий параметр).

let num1 = parseInt("**42**") // Виведе 42

let num2 = parseInt("**10**", **2**) // Виведе 2 (перетворення з двійкової системи числення)

let num3 = parseInt("**FF**", **16**) // Виведе 255 (перетворення з шістнадцяткової системи числення)

Якщо рядок не містить допустимого числа, функція parseInt поверне значення **NaN (**Not a Number**)**.

let num4 = parseInt("Hello") // Виведе NaN  
let num5 = parseInt("Hello22") // Виведе NaN  
let num6 = parseInt("22Hello") // Виведе 22  
let num7 = parseInt(" ") // Виведе NaN

Важливо зазначити, що функція parseInt також відкидає десяткову частину числа, якщо вона присутня.

let num8 = parseInt("3.14") // Виведе 3

Якщо початкове значення не string, воно спочатку буде перетворенго у string, а потім знову у число.  
Дані, які є меншими за 10 – 6 представляються у експоненціальній формі

let num9 = parseInt(0.000007) // (7 - 6) Виведе 0  
let num10 = parseInt(0.00000007) // (7 - 8) Теж саме, що parseInt(‘7e-8’). Тому виведе 7

**parsFloat –** функція яка використовується для перетворення рядка (string) на число з плаваючою крапокю

let num1 = parseFloat("3.14") // Виведе 3.14

let num2 = parseFloat("Hello") // Виведе NaN

let num3 = parseFloat("3.14abc") // Виведе 3.14

**Number() –** функція яка використовується для перетворення рядка (string) на число

let num1 = Number("30") // Виведе 30

let num2 = Number("30-fg") // Виведе NaN

let num3 = Number("fg-30") // Виведе NaN   
let num4 = Number(" ") // Виведе 0

**+ –** оператор, який перетворює рядок (string) на число

let num1 = + "30" // Виведе 30

let num2 = + "30-fg" // Виведе NaN

let num3 = + "fg-30" // Виведе NaN

**Неявне перетворення** рядкана число

let num1 = "30" \* 2 // Виведе 60 = Number("30") \* 2 = 30 \* 2

let num2 = " " \* 2 // Виведе 0 = Number("30") \* 2 = 0 \* 2

let num2 = "30-fg" \* 2 // Виведе NaN = Number("30") \* 2 = NaN \* 2

|  |  |  |
| --- | --- | --- |
| Тип даних | Приклад значення | Результат |
| Рядок | "42" | 42 |
| Рядок | "3.14" | 3.14 |
| Рядок | порожній рядок | 0 |
| Рядок | "ABC" | NaN |
| Булеве значення | true | 1 |
| Булеве значення | false | 0 |
| null | null | 0 |
| undefined | undefined | NaN |
| Об'єкт | { value: 42 } | NaN |
| Масив | [1, 2, 3] | NaN |

Метод toFixed()

**toFixed() –** є методом, який доступний для числових значень. Він використовується для форматування числа з плаваючою крапкою та задання кількості десяткових знаків після коми.

number.toFixed(digits)

* **digits** - кількість десяткових знаків, які мають бути виведені після коми. Може бути цілим числом від 0 до 20. (необов'язковий параметр)

let number = 3.14149

let newNumber = number.toFixed(**3**) // Виведе рядок з значенням "3.141"

**Важливо** **!** toFixed() повертає рядок, а не число.

let realNumber = parseFloat(newNumber) // Виведе 3.141

Метод isFinite()

**isFinite() –** є методом, який доступний для числових значень. Він икористовується для перевірки, чи число є кінцевим (не безкінечністю або не NaN). Якщо число не є безкінечністю (позитивною або від'ємною) та не є NaN, то isFinite() повертає true, інакше він повертає false

let num1 = 42

let num2 = Infinity

let num3 = NaN

console.log(isFinite(num1)) // Виведе true, тому що 42 - кінцеве число.

console.log(isFinite(num2)) // Виведе false, тому що Infinity - не є кінцевим числом.

console.log(isFinite(num3)) // Виведе false, тому що NaN - не є кінцевим числом.

Якщо потрібно перевірити, чи число не є безкінечністю (позитивною або від'ємною) або NaN, можна використати !isFinite(), який вертає true, якщо число є безкінечністю, і false, якщо число є кінцевим.

let num = 42

if (!isFinite(num)) {

console.log("Число є безкінечністю або NaN.")

} else {

console.log("Число є кінцевим числом.") // Виведе "Число є кінцевим числом."

}

Арифметичні вирази

**Арифметичний вираз** - це вираз, що складається з операторів та операндів, в якому виконуються арифметичні операції, результатом обчислення якого є число.

**Бінарні операції** - це операції, які виконуються між двома операндами.

|  |  |  |  |
| --- | --- | --- | --- |
| Оператор |  | Опис | Приклад |
| + | 12 | Додавання | 5 + 3 |
| - | 12 | Віднімання | 8 - 2 |
| \* | 13 | Множення | 4 \* 6 |
| / | 13 | Ділення | 10 / 2 |
| % | 13 | Остача від ділення (модуль) | 7 % 3 |
| > | 10 | Більше ніж | 5 > 2 |
| < | 10 | Менше ніж | 3 < 7 |
| >= | 10 | Більше або дорівнює | 4 >= 4 |
| <= | 10 | Менше або дорівнює | 6 <= 9 |
| === | 9 | Рівне | 2 === 2 |
| !== | 9 | Нерівне | 5 !== 3 |
| && | 5 | Логічне І | true && false |
| || | 4 | Логічне АБО | true || false |

**Унарні оператори** - це оператори, які працюють з одним операндом.  
Унарні оператори використовуються для здійснення різних операцій, таких як зміна знаку числа, інкремент або декремент числа, логічне заперечення та інші.

Унарні оператори можуть бути застосовані до чисел, рядків, змінних, об'єктів та інших типів даних, залежно від їх призначення.

| **Оператор** | **Опис** |
| --- | --- |
| **+** | **Унарний плюс**. Перетворює операнд на число. |
| **-** | **Унарний мінус**. Змінює знак числа. |
| **++** | Унарний **оператор інкремента**. Збільшує значення операнду на одиницю. |
| **--** | Унарний **оператор декремента**. Зменшує значення операнду на одиницю. |
| **!** | **Логічне заперечення**. Перетворює операнд на логічне значення та повертає його заперечення. |
| **typeof** | Повертає рядкове представлення типу операнду. |
| **delete** | Видаляє властивість об'єкту або елемент масиву. |
| **void** | Повертає undefined. |
| **~** | **Побітове заперечення**. Перетворює операнд на 32-бітне ціле число та повертає його заперечення. |



**Основні методи Math**

|  |  |
| --- | --- |
| Метод | Опис |
| Math.abs(x) | Повертає абсолютне значення числа **x**. |
| Math.ceil(x) | Округлює число **x** вгору до найближчого більшого цілого числа. |
| Math.floor(x) | Округлює число **x** вниз до найближчого меншого цілого числа. |
| Math.round(x) | Округлює число **x** до найближчого цілого числа (за правилами округлення). |
| Math.max(x1, x2, ...) | Повертає найбільше число серед **x1**, **x2**, ... |
| Math.min(x1, x2, ...) | Повертає найменше число серед **x1**, **x2**, ... |
| Math.pow(x, y) | Підносить число **x** до степеня **y**. |
| Math.sqrt(x) | Повертає квадратний корінь числа **x**. |
| Math.random() | Генерує випадкове дробове число в діапазоні від 0 (включно) до 1 (не включаючи). |
| Math.sin(x) | Повертає синус числа **x** (в радіанах). |
| Math.cos(x) | Повертає косинус числа **x** (в радіанах). |
| Math.tan(x) | Повертає тангенс числа **x** (в радіанах). |
| Math.log(x) | Повертає натуральний логарифм числа **x**. |
| Math.exp(x) | Повертає експоненту (число e, приблизно 2.71828) піднесену до степеня **x**. |
| Math.atan(x) | Повертає арктангенс числа **x**. |
| Math.acos(x) | Повертає арккосинус числа **x**. |
| Math.asin(x) | Повертає арксинус числа **x**. |
| Math.trunc(x) | Видаляє десяткову частину числа **x**, повертаючи тільки цілу частину без округлення. |
| Math.sign(x) | Повертає знак числа **x** як -1, 0 або 1, вказуючи на від'ємне, нульове або додатнє значення відповідно. |
| Math.log10(x) | Повертає логарифм числа **x** за основою 10. |
| Math.log2(x) | Повертає логарифм числа **x** за основою 2. |
| Math.cbrt(x) | Повертає кубічний корінь числа **x**. |
| Math.hypot(x1, x2, ...) | Повертає гіпотенузу для заданих сторін прямокутного трикутника зі сторонами **x1**, **x2**, ... (квадратний корінь суми квадратів). |

**Перетворення** різних типів даних **на рядок (стрінгу)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Тип даних | toString() | toFixed() | toPrecision() | toLocaleString() | JSON.stringify() | String() | Конкатенація |
| Number | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| Boolean | ✔ |  |  |  | ✔ | ✔ | ✔ |
| String | ✔ |  |  | ✔ | ✔ | ✔ | ✔ |
| Array | ✔ |  |  |  | ✔ | ✔ | ✔ |
| Object | ✔ |  |  |  | ✔ | ✔ | ✔ |

**Конкатенація** - процес об'єднання двох або більше рядків в один (або одного рядка та іншого типу даних).

* Використання оператора **"+"**

let result1 = 'Hello' **+** ' world!' // Конкатенація рядків. Виведе "Hello world!"  
let result2 = 12 + ‘33’ // Виведе "1233"  
let result3 = 12 + ‘ ’ // Виведе "12" = 12 + Number(" ") = 12 + 0   
let result3 = 12 + ‘ ’ \* 4 // Виведе 12 = 12 + Number(" ") \* 4 = 12 + 0 \* 4

* Використання методу **concat()**

let str1 = 'Hello'

let str2 = ' world!'

let result = str1**.concat(str2)** // Конкатенація рядків

console.log(result) // Виведе "Hello world!"

* Використання **шаблонних рядків** (Template literals)

var name = "John"

var age = 30

var str = `My name is **${name}** and I am **${age}** years old.`

console.log(str) // Виведе "My name is John and I am 30 years old."

* Метод **join():** Цей метод масиву об'єднує всі елементи масиву в один рядок, розділених заданим роздільником.

var arr = [1, 2, 3]

var str = arr**.join("-")**

console.log(str) // Виведе "1-2-3"

* Метод **toString()**

var num = 42

var str = num**.toString()**

console.log(str) // Виведе "42"

var bool = true

var str = bool**.toString()**

console.log(str) // Виведе "true"

Логічний тип

Логічний тип даних **Boolean** має всього два значення - **true** («істина») **false** («неправда», «невірно»)

**Перетворення** до логічного типу даних  **Boolean()**

**Boolean()** - виконується для визначення, чи є значення істинним (true) або неправдивим (false)

let a = 10

let var1 = Boolean(a) // true  
let var2 = !!a // true ! ( ! ( Boolean(a) = true ) = false ) = true

**!** - Логічне заперечення. Перетворює операнд на логічне значення та повертає його заперечення.

|  |  |
| --- | --- |
| Значення | Результат |
| false | false |
| Число 0 | false |
| Порожній рядок " " | false |
| null | false |
| undefined | false |
| NaN | false |
| -------------------------------------------------------------------------------------------------------------------- |  |
| true | true |
| Будь-яке число, відмінне від 0 (включаючи від'ємні і десяткові числа) -1, 1, 13,5 | true |
| Непорожній рядок “Hello” | true |
| Об'єкт {} | true |
| Масив [] | true |
| Будь-яка функція function(){} | true |

Логічні вирази

Логічний вираз, це вираз результат обчислення якого є **true** або **false**.

**Логічні вирази можуть містити:**

* **логічні константи (true/false**) - це фіксовані значення, які представляють логічний стан істинності або хибності.

let x = 5

if (x > 0)

console.log("x є додатним числом") // true

else

console.log("x не є додатним числом") // false

* **змінні логічного типу**

let isTrue = true

let isFalse = false

console.log(isTrue) // true

console.log(isFalse) // false

* **оператори порівняння -** використовуються для порівняння значень і повертають логічне значення true або false залежно від результату порівняння.

let a = 5, b = 10

|  |  |  |  |
| --- | --- | --- | --- |
| console.log(a == b) | 9 | false | Рівність |
| console.log(a === b) | 9 | false | Строга рівність |
| console.log(a != b) | 9 | true | Нерівність |
| console.log(a !== b) | 9 | true | Строга нерівність |
| console.log(a > b) | 10 | false | Більше |
| console.log(a < b) | 10 | true | Менше |
| console.log(a >= b) | 10 | false | Більше або рівне |
| console.log(a <= b) | 10 | true | Менше або рівне |

**== (рівність)**: Порівнює два значення на рівність, якщо необхідно **перетворюючи типи даних у Number**. Наприклад - 1 == '1' буде true, оскільки значення рівні після перетворення типів.

**=== (строга рівність)**: Порівнює два значення на рівність, без перетворення типів. Воно порівнює значення і тип даних.

Наприклад - 1 === '1' буде false, оскільки значення рівні, але типи даних різні.

**!= (нерівність)** Порівнює два значення на нерівність, якщо потрібно **перетворює типи даних у Number**. Наприклад - 1 != '1' буде false, оскільки значення рівні після перетворення типів.

**!== (строга нерівність)**: Порівнює два значення на нерівність, без перетворення типів. Воно порівнює значення і тип даних.

Наприклад - 1 !== '1' буде true, оскільки значення рівні, але типи даних різні, тому вони є строго нерівними.

Строгі оператори рівності **===** та **!==**  часто вважаються безпечнішими, оскільки вони не здійснюють автоматичне перетворення типів, дозволяючи точнішу перевірку значень і типів даних

Операції з логічними виразами

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x | y | x || y 4 | x && y 5 | ^ 7 | !x 15 |
| false | false | false | false | false | true |
| false | true | true | false | true | true |
| true | false | true | false | true | false |
| true | true | true | true | false | false |

* Логічне і (&&): Повертає true, якщо обидва операнди є true, інакше повертає false.

var x = 5

var y = 10

var z = 15

console.log(x < y && y < z) // Виведе true - оскільки обидва умови виконуються

console.log(x < y && y > z) // Виведе false - оскільки друга умова не виконується

* Логічне АБО (||): Повертає true, якщо хоча б один з операндів є true, інакше повертає false.

var x = 5

var y = 10

var z = 15

console.log(x < y || y < z) // Виведе true - оскільки хоча б одна з умов виконується

console.log(x > y || y > z) // Виведе false - оскільки обидві умови не виконуються

* Логічне НЕ (!): Інвертує значення операнду (true перетворюється на false і навпаки).

var x = 5

var y = 10

console.log(!(x < y)) // Виведе false - оскільки x < y є true, а !true = false

console.log(!(x > y)) // Виведе true - оскільки x > y є false, а !false = true

Алгоритми з розгалуженням

Алгоритми з розгалуженням - дозволяють приймати рішення залежно від умов, що задовольняють чи не задовольняють певній умові. Зазвичай алгоритми з розгалуженням включають в себе інструкції, які дозволяють виконувати певні дії в залежності від вхідних даних або стану програми.

Умовний оператор **if-else**

**if** (умова) {

// код, який виконається, якщо умова істинна

} **else** {

// код, який виконається, якщо умова не істинна

Цей оператор дозволяє виконати певний блок коду, якщо задана умова є істинною (true). Умовний оператор може бути розширений використанням else, else if, а також вкладеними умовами.

Тернарний оператор

Тернарний оператор (тернарний вираз): Це короткий спосіб виразити умовний оператор if-else у вигляді одного виразу. Він має такий формат:

let результат = (умова) ? 'Код який виконається якщо умова істинна' : 'Або якщо умова хибна';

let age = 20

let message = **(**age >= 18**) ? "**Ви повнолітній**" : "**Ви неповнолітній**"**

console.log(message) // Виведе “Ви повнолітній”

У цьому прикладі, змінна age містить значення 20. Умова (age >= 18) перевіряє, чи age більше або дорівнює 18. Якщо це вірно, то змінна message отримує значення "Ви повнолітній". У протилежному випадку, коли умова є хибною, message отримує значення "Ви неповнолітній".

Умовний оператор **switch**

Цей оператор порівнює значення виразу з усіма значенями (case). Якщо при порівнянні буде знайдено відповідність, виконується код, всередині case. Якщо жоден з випадків не збігається, виконується блок коду, що розташований всередині **default**.

let medal = 2

let medalName

switch (medal) {

case **1**:

medalName = "Золота"

break

case **2**:

medalName = "Срібна"

break

case **3**:

medalName = "Бронзова"

break

**default**:

medalName = "Хибні дані"

break

}

console.log("Медаль " + medalName)

! Важливо   
В case можна вказувати лише конкретні значення або константи, **вирази** вказувати **не можна**

Короткий цикл обчислень

Короткий цикл обчислень дозволяє програмі пропустити зайві оцінки умов і покращити продуктивність, вибираючи перше "істинне" значення або зупиняючись на першій "істинній" умові.

Логічний оператор “АБО” **|| 4**

У цьому рішенні, використовуючи логічний оператор "АБО" (||), ми перевіряємо кожну змінну від a до d на "істинність" значення (**true**)

var a = ‘’

var b = null

var c = 'World'

var d = undefined

var result = a || b || c || d || 'No value found'

console.log(result) // 'World'

Результатом обчислення буде значення виразу, булеве значення якого = **true** (усі наступні вирази не обчислюються).

У цьому прикладі, перша умова **a** перевіряє, яким є значення **a** при перетворенні до логічного типу

даних Boolean - **true** або **false**.

Значення **a** =’’ - це порожній рядок – воно означає що **а** – **false**, умова не виконується .

Таким же чином перевіряється кожна наступна умова b || c || d. Як тільки якась умова матиме "істинне" значення **true**, змінній result присвоїться її значення.

Значення **b** = null (**false) –** умова теж не виконується.

Значення **с** = 'World' – непорожній рядок – означає що **с** – **true.**

Наступна умова **d** не оцінюється.

Логічний оператор “ i ” && **5**

У цьому рішенні, використовуючи логічний оператор "І" (&&), ми перевіряємо кожну змінну від a до d на "хибне" значення (**false**).

var a = true

var b = 'World'

var c = undefined

var d = 0

var result = a && b && c && d

console.log(result) // undefined

Результатом обчислення буде значення виразу, якому еквівалентне **false** (усі наступні вирази не обчислюються).

Значення **a** = true - це означає що **а** присвоєно значення – **true**, умова не виконується .

Значення **b** = 'World' – непорожній рядок – означає що **b** – **true.**

Значення **с** = undefined – означає що змінна **с** не має визначеного значення, отже– **false.**

Наступна умова **d** не оцінюється.

Оператор нульового злиття “ Nullish Coalescing Operator “ ?? **4**

У цьому рішенні, використовується оператор ?? (Nullish Coalescing Operator)

var a = null

var b = undefined

var c = 'Hello'

var d = 'World'

var result = a ?? b ?? c ?? d

console.log(result) // 'Hello'

Результатом обчислення буде значення виразу, який не є **null** або **undefined** (усі інші вирази праворуч не обчислюються).

Змінній **result** присвоїться значення змінної c = **'Hello'**, так як це перша умова яка не є **null** або **undefined**.

Оператор умовного присвоєння ??= **2**

Оператор умовного присвоєння **??=** може бути використаний в короткому циклі обчислень для швидкої перевірки та присвоєння значень змінним, якщо вони є null або undefined.

var a = null

var b = 'Hello'

var c = undefined

var d = 'World'

a ??= 'Default Value'

b ??= 'Default Value'

c ??= 'Default Value'

d ??= 'Default Value'

console.log(a) // 'Default Value'

console.log(b) // 'Hello' (значення не змінилося)

console.log(c) // 'Default Value'

console.log(d) // 'World' (значення не змінилося)

Цикли

**Цикл** - конструкція мови, яка дозволяє виконувати певний блок коду або набір інструкцій повторно. Цикли дозволяють автоматизувати повторні завдання, виконувати обробку масивів, працювати з колекціями даних і багато іншого.

Цикл з параметром **for**

**for** - використовується для повторення блоку коду певну кількість разів або до досягнення певної умови.

for (ініціалізація; умова; ітерація) {

// блок коду, який виконується

}

**ініціалізація** - вказує початкове значення змінної, яка контролює цикл.

**умова** - це булевий вираз, який перевіряється перед кожною ітерацією. Якщо умова є true, то блок коду виконується. Якщо умова є false, то цикл завершується.

**ітерація** - визначає дії, які виконуються після кожної ітерації циклу. Зазвичай вона використовується для зміни значення змінної, яка контролює цикл.

Цикл з параметром **while** (цикл з передумовою)

**while** - Використовується для виконання блоку коду, доки задана умова є істинною. Умова перевіряється перед кожною ітерацією циклу.

while (умова) {

// блок коду, який виконується

}

**умова** - це булевий вираз, який перевіряється перед кожною ітерацією. Якщо умова є true, то блок коду виконується. Якщо умова є false, то виконання циклу припиняється, і керування передається наступній інструкції після циклу.

let i = 0

while (i < 5) {

console.log(i)

i++

}

У цьому прикладі блок коду виконується, доки змінна **i** менше 5. Змінна **i** початково має значення 0. Після кожної ітерації значення i збільшується на 1. Результатом будуть числа від 0 до 4, виведені в консоль.

Цикл з параметром **do … while** (цикл з післяумовою)

**do...while** - тип циклу, який виконує блок коду перед перевіркою умови. Після виконання блоку коду умова перевіряється, і якщо вона є істинною, цикл продовжується.

do {

// блок коду, який виконується

} while (умова)

let i = 0

do {

console.log(i)

i++

} while (i < 5)

У цьому прикладі блок коду виконується спочатку, незалежно від умови. Потім, після кожної ітерації, перевіряється умова **i < 5**. Якщо умова є **true**, цикл продовжується і блок коду виконується знову. Цикл завершується, коли умова стає **false**. Результатом будуть числа від 0 до 4, виведені в консоль.

Оператор **break**

Використовується для негайного виходу з циклу або переривання виконання блоку коду. Коли виконується оператор **break**, керування передається наступній інструкції після циклу або блоку коду, що містить break.

Оператор **break** може бути використаний у циклах (**for, while, do...while**) та блоках коду, таких як **switch**.

let i = 0

while (i < 5) {

console.log(i)

i++

if (i === 3)

break

}

У цьому прикладі, коли i досягає значення 3, виконується оператор **break**, і цикл **while** припиняє свою роботу. Результатом будуть числа 0, 1 і 2, виведені в консоль.

Оператор **break** з міткою

Використовується для виходу з зовнішнього циклу або блока коду, коли є вкладені цикли або блоки коду.

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

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

if (i === 1 && j === 1)

break out

console.log(`i: ${i}, j: ${j}`)

}

}

У цьому прикладі ми маємо вкладений цикл **for**, який виконується для значень **i** від 0 до 2 і **j** від 0 до 2. Умова **if** перевіряє, чи **i** дорівнює 1 і **j** дорівнює 1. Якщо ця умова є істинною, то за допомогою оператора **break out** виконання циклу припиняється і вихід здійснюється зовнішньої мітки **out**. Це означає, що весь вкладений цикл припиняється і керування переходить на наступну інструкцію після зовнішнього циклу.

Оператор **continue**

При використанні оператора **continue** в коді, виконання поточної ітерації циклу припиняється, і керування переходить до наступної ітерації циклу.

let i = 0

while (i < 5) {

i++

if (i % 2 === 0)

continue

console.log(i)

}

У цьому прикладі, якщо **i** є парним числом, оператор **continue** виконується, і поточна ітерація циклу припиняється. Керування переходить до наступної ітерації циклу, і парне число пропускається. Результатом будуть непарні числа 1, 3 і 5, виведені в консоль.

Функція

**Функція** є блоком коду, який виконує певну задачу або обчислення. Вона складається з ключового слова **function**, **назви функції**, **списку формальних параметрів** у круглих дужках, **тіла функції** і можливого **return** повернення значення.

* **Оголошення функції**: Використовується ключове слово **function**, за яким йде назва функції.

**function myFunction**(a, b) {

// код функції

}

* **Формальні параметри**: Функції можуть мати вхідні параметри, які приймають значення, коли функція викликається. Вони вказуються у круглих дужках, розділені комами.

function myFunction**(a, b)** {

sum = a + b

return sum

}

* **Тіло функції**: Це блок коду, який виконується при виклику функції. Він містить інструкції та обчислення, які виконуються при виклику функції.

function myFunction (a, b) {

**sum = a + b**

return sum

}

* **Повернення значення**: Функція може повертати значення, за допомогою ключового слова return. Це значення може бути використане у виразах або збережене в змінну.

function myFunction (a, b) {

sum = a + b

**return sum**

}

Функції можуть бути визначені і викликані в будь-якому місці програми. Їх можна передавати як аргументи в інші функції і повертати як значення. Функції є потужним інструментом для структурування та повторного використання коду.

Задання значень формальних параметрів за замовчуванням (default values)

При оголошенні функції можна встановити початкові значення для формальних параметрів, які використовуються у випадку, якщо аргументи не передаються під час виклику функції.

function greet(firstName **=** ‘’Anonymous’’, lastName **=** ‘ ‘) {

return (`Hello, ${ firstName} ${ lastName}!`)

}

greet() // Виведе "Hello, Anonymous !"

greet(', John') // Виведе "Hello, Anonymous John!"

Функції - стрілки

Надають зручний та короткий спосіб оголошення функцій.

Обидві функції виконують одну і ту саму задачу - обчислення суми двох чисел. Однак, є кілька відмінностей між ними:

Звичайна функція:

function sum(a, b) {

return a + b

}

console.log(sum(2, 3)) // Виведе 5

Функція-стрілка:

let sum = (a, b) **=>** a + b

console.log(sum(2, 3)) // Виведе 5

* **Синтаксис**  
  Звичайна функція використовує ключове слово function, вказує ім'я функції та має блок коду з вказанням return для повернення значення.

Функція-стрілка має коротший синтаксис з використанням стрілки (**=>**), без ключового слова **function** та без вказання блоку коду, якщо вона повертає значення без додаткових обчислень.

* **Контекст this**  
  У звичайних функціях контекст this залежить від того, як саме викликано функцію (наприклад, через об'єкт або через конструктор). У функціях-стрілках відсутній власний контекст this та вони успадковують його від зовнішнього середовища.
* **Виклик функції**Звичайні функції можуть бути викликані як звичайні функції через ім'я функції, а також використовуватись як конструктори за допомогою ключового слова new. Функції-стрілки не можуть бути викликані як конструктори та не мають свого власного this, тому використання new з функціями-стрілками видасть помилку.
* **Коротший код**  
  Функції-стрілки мають коротший синтаксис, що дозволяє скоротити кількість коду, особливо якщо функція складається з одного виразу без фігурних дужок.
* **return** – не обов’язковий
* Якщо тіло функції-стрілки складається з одного виразу без фігурних дужок.

const sum = (a, b) => a + b

У цьому випадку результат виразу a + b автоматично повертається з функції-стрілки.

* Якщо тіло функції-стрілки складається з блоку коду (з фігурними дужками), то необхідно вказати оператор return, якщо потрібно явно повернути значення.

const multiply = (a, b) => {

const result = a \* b

return result

}

У цьому випадку оператор return використовується для повернення значення result.

Масиви

**Масив (Array)** це нумерована послідовність елементів даних, яка може містити значення різних типів - числа, рядки, об'єкти, функції а також інші масиви.

**Основні риси масиву** в JavaScript:

1. **Упорядкованість**  
   Елементи масиву зберігаються послідовно в пам’яті один за одним.  
   Кожен окремий елемент масиву має свою позицію в масиві (**індекс**).  
   Доступ (звернення) до них здійснюється за допомогою **індексів**.   
   Індекси починаються з 0, тобто перший елемент масиву має індекс 0, другий - індекс 1 і так далі. Завдяки цьому, елементи в масиві вважаються згрупованими.
2. **Динамічність**  
   Розмір масиву може змінюватись під час виконання програми. Можна додавати нові елементи, видаляти існуючі або змінювати вже наявні.
3. **Різні типи даних**  
   Масив може містити елементи різних типів даних (які можуть повторюватись), таких як числа, рядки, булеві значення, об'єкти, функції та інші масиви. Це робить масиви в JavaScript універсальними для зберігання та обробки різноманітних даних.
4. **Методи та властивості**  
   Масиви мають багато вбудованих методів, які дозволяють виконувати різноманітні операції з масивами, такі як додавання/видалення елементів, сортування, пошук, фільтрація і багато іншого. Також є різні властивості, які надають інформацію про масив, наприклад, його довжину (**length**).

const numbers = [1, 2, 3, 4, 5] // Масив чисел

const fruits = ['apple', 'banana', 'orange'] // Масив рядків

const mixedArray = [1, 'hello', true, { name: 'John' }] // Масив з різними типами даних

**Розрізняють наступні види масивів**

1. **Одновимірні.**   
   В одновимірних масивах елементи формуються в ряд. Доступ до елементів масиву здійснюється шляхом вказання одного індексу (позиції);
2. **Багатовимірні.**   
   Тут елементи сформовані за двома, трьома і більше вимірами. Кожен вимір це також ряд елементів. Доступ до елементів багатовимірних масивів здійснюється на основі двох, трьох і більше індексів.

**Оголошення (створення) одновимірного масиву**

1. Створення масиву за допомогою оператора **new** з конструктором масиву **Array**

const array = **new** Array(1, 2, 3, 4, 5)

Використання **new Array()** може бути корисним у випадках, **коли потрібно виділити пам’ять для елементів масиву наперед** (створити масив з обмеженою кількістю пустих елементів) або зазначити лише один елемент.

(спочатку елементи є порожніми (**undefined**)

let studentNumber = parseInt(prompt('Введіть кількість учнів'))

let studentAges = **new Array**(studentNumber)

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

    let studentAge = parseInt(prompt(`Введіть вік ${i}-го учня`))

    studentAges[i] = studentAge

}

1. Літеральне створення масиву **[]**

const array = **[**1, 2, 3, 4, 5**]**

Літеральне створення масиву використовує квадратні дужки (**[]**) і список значень, розділених комами.   
Виділення пам'яті для елементів масиву (розмір масиву) при літеральному створенні масиву відбувається під час виконання програми, коли доходить черга до виконання відповідного коду.

let studentNumber = parseInt(prompt('Введіть кількість учнів'))

let studentAges = **[]**

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

    let studentAge = parseInt(prompt(`Введіть вік ${i}-го учня`))

    studentAges.push(studentAge)

}

Доступ до елементів масиву. Властивості індексу масиву. Додавання елементів до масиву.

Після оголошення масиву, можна цей масив використовувати або змінювати. Доступ до окремого елементу масиву виконується з допомогою квадратних дужок **[]**

ArrayName[index]

ArrayName – ім’я масиву;

index – позиція в масиві. Це може бути число, рядок, символ тощо.

Після оголошення масиву з деяким набором елементів, у цьому масиві значення index формуються у зростаючому порядку, починаючи з 0 (0, 1, 2, …). Однак, мова JavaScript допускає використовувати в якості index будь-який об’єкт. Цим об’єктом може бути рядок, число з плаваючою комою, ціле число, символ тощо.

ArrayName['+'] = 2.88

В цьому прикладі створюється масив з одним елементом, зі значенням 2.88. Його індексом є значення ‘+’

var c = ArrayName['+']

Змінна c буде мати значення 2.88.

Якщо змінній неіснуючий індекс масиву, то результатом буде значення undefined

var A = [ 1, 2, 5]

var item = A[15] // item = undefined

**Визначення розміру масиву.** Властивість**length**

Щоб визначити розмір масиву використовується властивість length, яка повертає ціле число, що рівне кількості елементів масиву. Звертання до властивості для масиву A може бути таким.

A.length

Властивість **length** базується на найвищому значенні індексу занятого елементу в масиві, а не на кількості фактично створених елементів.



Об’єкт **arguments** (псевдо-масив)

**arguments** - це локальний об'єкт, який використовується всередині функцій для отримання доступу до переданих під час виклику аргументів. Доступ до індексів, а також до кількості переданих аргументів.

function sum() { // параметри відсутні

let total = 0

for (let i = 0; i < arguments.length; i++) total += arguments[i]

return total

}

console.log(sum(1, 2, 3)) // Виведе 6, тут передано три аргументи

console.log(sum(10, 20, 30, 40)) // Виведе 100, тут передано чотири аргументи

-------------------------------------------------------------------------------------------------------------

function getFinalSalary() { // параметри відсутні

return ((arguments[0] + arguments[1] )\* (1 – (arguments[2] / 100))

}

console.log(getFinalSalary()) // Виведе розмір заробітньої плати ((ставка + премія) - податок)

**arguments** є об'єктом, а не масивом, тому він не має доступу до методів масивів.

Важливо також зазначити, що з ES6 (ECMAScript 2015) введено стрілкові функції, які не мають власного об'єкта arguments. Якщо потрібно отримати аргументи в стрілковій функції, потрібно використовувати оператор розширення **spread** (...) для отримання параметрів у вигляді масиву.

const sumArrow = (...args) => {

return args.reduce((total, current) => total + current, 0)

}

const result = sumArrow(2, 4, 6, 8)

console.log(result) // Виведе 20

Оператор **spread** ( **…** )

Оператор **spread** (розпилювання) - це оператор, який дозволяє розподілити (розпилити) елементи масиву або властивості об'єкту в інший масив або об'єкт.

* **Розпилювання масиву:**

const numbers = [11, 22, 33];

const newArray = [...numbers, 4, 5];

console.log(newArray); // Виведе [11, 22, 33, 4, 5]

В цьому прикладі оператор **spread** розпилює елементи масиву **numbers** в новий масив **newArray**. Результатом буде масив, що складається з елементів масиву **numbers** та додаткових значень [**4, 5**].

* **Розпилювання об'єкта:**

const person = { name: 'John', age: 30 };

const newPerson = { ...person, city: 'New York' };

console.log(newPerson); // Виведе { name: 'John', age: 30, city: 'New York' }

Тут оператор spread розпилює властивості об'єкта person в новий об'єкт newPerson. Результатом буде об'єкт, який містить всі властивості об'єкта person та додаткову властивість city.

* **Передача аргументів функції:**

const numbers = [1, 2, 3, 4, 5];

const maxNumber = Math.max(...numbers);

console.log(maxNumber); // Виведе 5

Тут оператор spread розпилює елементи масиву numbers як окремі аргументи для функції Math.max(), що дозволяє знайти найбільше число у масиві.

**Деструктуризація масиву**

**Деструктуризація масиву** - це спосіб отримання окремих елементів масиву, у вигляді змінних. За допомогою деструктуризації масиву, можна присвоїти значення змінним на основі елементів масиву.

Кількість змінних, які ви використовуються при деструктуризації об'єкту, може бути меншою або більшою, ніж кількість елементів масиву.  
В цьому випадку будуть або, пропущені непотрібні елементи масиву, або змінні отримають значення **undefined.**

|  |  |
| --- | --- |
| Вар. 1  const numbers = **[**1, 2, 3, 4, 5**]**  const **[**a, b**]** = numbers  console.log(a) // Виведе 1  console.log(b) // Виведе 2 | Вар. 2  const numbers = **[**1, 2**]**  const **[**a, b, c, d**]** = numbers  console.log(a) // Виведе 1  console.log(b) // Виведе 2  console.log(c) // Виведе undefined  console.log(d) // Виведе undefined |

**Приклад повернення декількох значень з функції за допомогою масиву**

function getFunction(a, b) {

return [a+b , a\*b]

}

let result = getFunction(2, 3)

alert(“Sum =” + **result[0]** + “Prod =” + **result[1]**)

**Деструктуризація об’єкту**

**Деструктуризація об'єкту** - це спосіб отримання окремих властивостей об'єкту, у вигляді змінних. За допомогою деструктуризації об’єкта, можна присвоїти значення змінним на основі елементів об’єкта.

Кількість змінних, які ви використовуються при деструктуризації об'єкту, може бути більшою або меншою, ніж кількість властивостей об'єкту.

В цьому випадку будуть або, пропущені непотрібні елементи об’єкту, або змінні отримають значення **undefined.**

|  |  |
| --- | --- |
| Вар. 1  const person = {  name: 'John',  age: 30,  city: 'New York'  }  const { newName } = person  console.log(newName) // Виведе 'John' | Вар. 2  const person = {  name: 'John',  age: 30  }  const { newName, newAge, newCity } = person  console.log(newName) // Виведе 'John'  console.log(newAge) // Виведе 30  console.log(newCity) // Виведе undefined |

**Приклад повернення декількох значень з функції за допомогою об’єкта**

function getFunction(a, b) {

return [sum:a+b , prod:a\*b]

}

let result = getFunction(2, 3)

alert(“Sum =” + **result.sum** + “Prod =” + **result.prod**)

Метод **fill**

Метод **fill()** є одним із методів масиву. Він використовується для заповнення всіх елементів масиву фіксованим значенням.

Синтаксис методу **fill()**:

array.fill(value, start, end)

* **value** - Значення, яким будуть заповнені елементи масиву.
* **start** (необов'язковий) - Індекс початку заповнення. За замовчуванням починається з індексу 0.
* **end** (необов'язковий) - Індекс кінця заповнення. Елемент з індексом end **не включений**. За замовчуванням заповнюються всі елементи масиву.

let array = new Array(7)

array.fill(**0**) // Заповнення всіх елементів масиву значенням 0

console.log(array) // Виведе [0, 0, 0, 0, 0, 0, 0]

array.fill(**7**, **2**) // Заповнення масиву 7-ми починаючи з елементу з індексом 2 і до кінця масиву

console.log(array) // Виведе [0, 0, 7, 7, 7, 7, 7]

array.fill(**9**, **1**, **4**) // Заповнення масиву 9-ми починаючи з елементу з індексом 1 до 4 (**4 не включно!**)

console.log(array) // Виведе [0, 9, 9, 9, 7, 7, 7]

Метод **fill()** повертає змінений масив. Важливо враховувати, що він виконує заміну значень в масиві без створення нового масиву.

**Ітеровані об’єкти**

Ітеровані об'єкти (**ітерабельні об'єкти**) є об'єктами, які підтримують ітерацію або послідовний перебір їх елементів. Такі об’єкти дозволяють отримати доступ до кожного їх елемента по одному, використовуючи певні методи або використовуючи цикл.

Ітеровані об'єкти - масиви (**Array**), рядки (**String**), об'єкти **Set**, об'єкти **Map**, **arguments** (об'єкт, який представляє аргументи, передані в функцію) та інші.

Для роботи з ітерованими об'єктами можна використовувати цикли, такі як **for...of**, або методи, які надаються для цих об'єктів, такі як **forEach**, **map**, **filter**, **reduce** та інші.

// Масив

const array = [1, 2, 3, 4, 5];

for (const element of array) {

console.log(element) // Виводить кожен елемент масиву

}

// Рядок

const str = 'Hello';

for (const char of str) {

console.log(char) // Виводить кожен символ рядка

}

**Перегляд** елементів масиву за допомогою цикла **for … of**

Цикл **for ... of** використовується для перебору елементів ітерованого об'єкта **без їх зміни** (наприклад, масиву, рядка, об'єкта Set, об'єкта Map тощо) один за одним. Синтаксис циклу **for...of**:

const array = [1, 2, 3, 4, 5];

**for** (let element **of** array) { // Генеруємо цикл, який

// **створить змінну element**, та присвоюватиме їй

// по черзі значення кожного елементу масиву **array**

console.log(element) } // після цього, виведе кожну змінну

В цьому прикладі кожна змінна (по суті, значення кожного елемента масиву) буде виведено на консоль.

Цикл **for...of** автоматично обходить всі елементи ітерованого об'єкта, **не потребує вказівки індексів або довжини**, і працює з будь-яким об'єктом, який є ітерованим. Він є зручним і простим способом перебору елементів масиву або інших колекцій даних.

**Перегляд та зміна** елементів масиву за допомогою цикла **for** та умовного операторa **if**

let p = [1, 2, 3, 4, 5, 6, 7, 8]

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

**if** (p[i] > 4)

console.log(p[i])

**За допомогою** використання циклу **for** окрім перегляду, **можливе** також **внесення змін до елементів** масиву

**Перегляд та зміна** елементів масиву за допомогою цикла **forEach**

Цикл **forEach** є альтернативою циклу for, який **дозволяє переглядати** **та змінювати елементи масиву**. Особливістю цього циклу є те, що він автоматично переглядає всі елементи масиву без необхідності вказувати індекс.

array1.forEach**(**

(item, index, baseArr) => дії над елементом масиву

**)**

**item –** змінна, у яку поступово передаються (копіюються) елементи з масиву (з яким проводимо операції)

**index –** індекс поточного елемента (над яким зараз проводяться обчислення)

**baseArr –** змінна, яка містить посилання на масив (з яким проводимо операції)

Дано масив чисел. Елементи, які більші за 7, замінити на 0.

let array1 = [1, 2, 4, 6, 7, 9, 10, 12]

array1.forEach(

(element, index, baseArr) => {

if (element > 7)

baseArr[index] = 0 // рівносильно array1[index] = 0

}

)

console.log(array1);

**Створення** масиву на основі елементів (**які змінювати НЕможна**) іншого ітерованого об'єкта

Array.from

…spread

Обидва методи дозволяють зробити **поверхневу** копію простих масивів. Для масивів, які містять в собі підмасиви (або масиви з об'єктів), такі методи **не підходять!**

**Створення** масиву на основі елементів (**які змінювати НЕможна**) іншого ітерованого об'єкта, з додатковим перетворенням елементів

«**Array.from**» Створення масиву на основі елементів **ітерованого об'єкта**

const arr1 = [1,2,3,4,5,6]

const arr2 = **Array.from**( arr1, (element, index) => element \* 3)

arr2 = [3,6,9,12,15,18]

«**map»**  На основі елементів масиву, повертає новий масив, з такою ж кількістю елементів

const arr1 = [1,2,3,4,5,6]

const arr2 = **arr1.map** ((element, index, baseArray) => element \* 3)

// const arr2 = **arr1.map** (element => element \* 3)

// якщо індексі посилання на базовий масив не використовуються, їх можна не писати

arr2 = [3,6,9,12,15,18]

Доступ до елементів масиву

Доступ за індексом починаючи від **лівого** краю масиву. Індекс вказуємо у **квадратних** дужках

let arr = [10, 20, 30, 40, 50]

let newArr = arr [2] // x = 30

«.**at»** Доступ за індексом починаючи від **правого** краю масиву (від -1). Індекс вказуємо у **круглих** дужках

-5 -4 -3 -2 -1

let arr = [10, 20, 30, 40, 50]

let x = arr**.at(-**2**)** // x = 40

Методи «.**push**»та«.**pop**». **Додавання** та **вилучення** елементів з **кінця** масиву.

«**.push**» Додавання нового значення в кінець масиву

let arr = [10, 20, 30, 40, 50]

arr**.push**(60) // arr = [10, 20, 30, 40, 50, 60]

«**.pop**» Видалення та повернення останнього елемента з масиву. Як результат ми отримаємо видалений елемент, який можемо використати за потреби.

let arr = [10, 20, 30, 40, 50]

let newArray = arr.pop()

// newArray = 50

// arr = [10, 20, 30, 40]

«.**length**» Якщо length присвоєно значення, яке є меншим за довжину масиву, то масив зменшується до вказаної кількості елементів.

let arr = [10, 20, 30, 40, 50]

arr.length = 3

// arr = [10, 20, 30]

Методи «**.unshift**» та«.**shift**». **Додавання** та **вилучення** елементів з **початку** масиву.

«**.unshift**» Додавання нового значення у початок масиву

let arr = [10, 20, 30]

arr**.unshift** (40)

// arr = [40, 10, 20, 30]

«**.shift**» Видалення та повернення першого елемента з масиву. Як результат ми отримаємо видалений елемент, який можемо використати за потреби.

0 1 2 3 4

let arr = [10, 20, 30, 40, 50]

let newArray = arr.**shift**()

// newArray = 10

// arr = [20, 30, 40]

**\*\*\*** Процес додавання та видалення першого елементу масиву може бути **ресурсозатратним та займати багато часу**.  
Якщо є можливість не викорстовувати **unshift** та **shift,** то не використовуємо!!!

Дії з масивом (при додаванні елемента у початок):

1. створено новий елемент в кінці масиву

arr = [10, 20, 30, ]

1. переміщення на одну позицію вправо

arr = [10, 20, ,30]

arr = [10, ,20, 30]

arr = [ ,10, 20, 30]

1. у першу позицію записуємо нове значення

arr = [40, 10, 20, 30]

Метод «**.splice**». **Додавання** та **вилучення** елементів з масиву у довільній позиції.

«**.splice**» Додавання (або видалення та повернення) елемента.

arrayName**.splice**( індекс елемента, скільки елементів видалити, перечислюємо нові значення)

--------------------------- Приклад 1 ------------------------------------------------

0 1 2 3 4

let arr = [10, 20, 30, 40, 50]

arr.splice (1, 0, 80, 90, 100) // arr = [10, 80, 90, 100, 20, 30, 40, 50]

--------------------------- Приклад 2 ------------------------------------------------

0 1 2 3 4 5

let arr = [10, 80, 20, 30, 40, 50]

let newArray = arr.splice(3,2)

// arr = [10, 80, 20, 50]

// newArr = [30, 40]

**\*\*\*** Процес додавання та видалення елеменів масиву може бути **ресурсозатратним та займати багато часу**.  
Якщо є можливість не викорстовувати **splice,** то не використовуємо!!!

---------------------------------- Задача ------------------------------------------------

Дамно масив років. Після кожного значення 2010 додати 2015.

      let years = [1999, 2011, 2000, 2020, 2010, 2019, 2023, 2010, 2018]

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

        if (years[i] === 2010) {

          years.splice(i + 1, 0, 2015)

          i++

        }

      }

      document.write(years) // years = [1999, 2011, 2000, 2020, 2010, 2015, 2019, 2023, 2010, 2015, 2018]

Метод «**delete**». **Видалення елементів** з масиву у довільній позиції.

«**delete**» Видалення значення елемента, при цьому сам елемент залишається і дорівнює ***undefined***

0 1 2 3 4

let arr = [10, 20, 30, 40, 50]

delete arr[3] // arr = [10, 20, 30, *undefined* ,50]

\*\*\* Краще використовувати **splice**!!!

Метод «**.slice**». **Копіювання фрагментів** масиву.

«**.slice**» Використовується для **копіювання** всього масиву або **чистини масиву**.

arrayName**.slice() -** весь масив скопіюється (аналог **spread**)

// newArray = arr.slice() | newArray = [...arr]

arrayName**.slice**( початковий індекс масиву, кінцевий індекс (не включаючи) ) - частина масиву

0 1 2 3 4

let arr = [10, 20, 30, 40, 50]

let newArr = arr.slice(2, 4) // newArr = [30, 40]

Методи «.**indexOf**» та«.**lastIndexOf**».Пошук **індексів** елементів масиву за вказаним значенням.

«**.indexOf**» Поверне індекс **першого зліва** елемента значення якого = вказаному

arrayName**.indexOf**(пошукове значення, початковий індекс пошуку)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

let arr = [17, 19, 23, 45, 42, 17, 23, 45, 42, 62, 14, 67, 43, 25, 75, 19, 20, 21, 38, 42]

let a = arr.indexOf(45) // 3

let b = arr.indexOf(23) // 2

let c = arr.indexOf(42,10) // 19

let d = arr.indexOf(23,10) // -1 (якщо такого елемента немає)

let e = arr.indexOf(23,b+1) // 6

«**.lastIndexOf**» Поверне індекс **першого зправа** елемента значення якого = вказаному

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

let arr = [17, 19, 23, 45, 42, 17, 23, 45, 42, 62, 14, 67, 43, 25, 75, 19, 20, 21, 38, 42]

let a = arr.lastIndexOf(45) // 7

let b = arr.lastIndexOf(23) // 6

Методи «**.findIndex**» та «.**findLastIndex**». Пошук **індексів** елементів маисву за вказаною умовою.

«**.findIndex**» Поверне індекс **першого** елемента (починаючи **зліва**), значення якого задовільняє умові.

arrayName**.findIndex(**(element, index, basseArray) => умова )

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

let arr = [17, 19, 23, 45, 42, 17, 23, 45, 42, 62, 14, 67, 43, 25, 75, 19, 20, 21, 38, 42]

let b = arr.findIndex(el => el >= 60) // 9

«**.findLastIndex**» Поверне індекс **першого** елемента (починаючи **зправа**), значення якого задовільняє умові.

arrayName**.findLastIndex(**(element, index, baseArray) => умова )

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

let arr = [17, 19, 23, 45, 42, 17, 23, 45, 42, 62, 14, 67, 43, 25, 75, 19, 20, 21, 38, 42]

let b = arr.findIndex(el => el >= 60) // 14

Методи «**.find**» «.**findLast**» «.**includes**». Пошук **елементів** масиву за вказаним значенням.

«**.find**» Поверне **перший** **елемент** (починаючи **зліва**), значення якого задовільняє умові.

arrayName**.find(**(element, index, baseArray) => умова )

let arr = [17, 19, 23, 45, 42, 17, 23, 45, 42, 62, 14, 67, 43, 25, 75, 19, 20, 21, 38, 42]

let newArray = arr.**find**(el => el >= 40) // 45

let newArray = arr.**find**(el => el >= 100) // *undefined*

«**.findLast**» Поверне **перший** **елемент** (починаючи **зправа**), значення якого задовільняє умові.

arrayName**.find(**(element, index, baseArray) => умова )

let arr = [17, 19, 23, 45, 42, 17, 23, 45, 42, 62, 14, 67, 43, 25, 75, 19, 20, 21, 38, 42]

let newArray = arr.**find**(el => el >= 40) // 45

let newArray = arr.**find**(el => el >= 100) // *undefined*

«**.includes**» Дозволяє визначити чи є **елемент** у масиві.

let arr = [17, 19, 23, 45, 42, 17, 23, 45, 42, 62, 14, 67, 43, 25, 75, 19, 20, 21, 38, 42]

arr.**includes**(67) // true

arr.**includes**(100) // false

Метод «**.reducе**» Обчислення агрегованого (акамулюючого) значення на основі елементів масиву.

arrayName**.reduce(**(початкове значення, element, index, baseArray) =>

умова,чому дорівнює початкове значення)

---------------------- Знайти суму парних елементів -------------------

let arr = [13, 9, 5, 13, 17, 16, 18, 19, 21, 11, 7, 4]

let sum = arr.reduce(

(prevSum, el) => (el % 2 === 0 ? prevSum + el : prevSum), 0

)

---------------------- Знайти добуток від’ємних елементів -------------------

let arr = [10, -9, 5, 13, 17, 16, 18, 19, 21, 11, 7, -4]

let prod = arr.reduce(

(prevProd, el) => (el < 0 ? prevProd \* el : prevProd), 1

)

---------------------- Знайти кількість 7 -------------------

let arr = [10, -9, 5, 13, 7, 16, 18, 7, 21, 11, 7, -4]

let count = arr.reduce(

(prevCount, el) => (el === 7 ? prevCount + 1 : prevCount), 0

)

---------------------- Знайти максимальний елемент -------------------

let arr = [10, -9, 5, 13, 7, 16, 18, 7, 21, 11, 7, -4]

let max = arr.reduce(

(prevMax, el) => (el > prevMax ? el : prevMax)

)

// якщо не вказувати початкове значення для prevMax, тоді prevMax = arr[0]

// теж саме - let max = Math.max(…arr)

---------------------- Знайти максимальний елемент серед від'ємних -------------------

let arr = [10, -9, 5, 13, 7, 16, 18, 7, 21, 11, 7, -4]

let min = arr.reduce(

(prevMin, el) => ((el < 0 && el > prevMin) ? el : prevMin), -infinity

)

Метод «**.filter**» Фільтрація елементів масиву.

«**.filter**» Повертає новий масив елементів, які задовільняють заданій умові. Якщо тих не має, то масив буде порожнім.

arrayName**.filter(**(element, index, baseArray) => умова )

-------------- Сформувати покази тільки доданих значень -------

let arr = [-10, 3, -21, 20, 18]

let positiveNum = arr.filter((el) => el > 0)

-------------- Сформувати масив імен які починаються на літери – A S J P -------

let arr = [Anna, Elena, Petr, Denys, Sasha, Jenya]

let check = [“A”, “S”, “J”, “P”]

let newArr = arr.filter(el => check.include(el.[0]))

Методи «**.every**» «**.some**» Перевірка елементів усього масиву на відповідність деякій умові

«**.every**» Повертає **true,** якщо усі елементи задовільняють заданій умові.

arrayName**.every (**(element, index, baseArray) => умова )

-------------- Визначити чи є він хорошистом (усі оцінки більші або рівні 7) -------

let scores = [8, 9, 12, 4, 7, 1, 10, 8, 9, 2]

      if (scores.every((score) => score >= 7)) document.write('Yes')

      else document.write('No')

«**.some**» Повертає **true,** якщо хочаб один елемент задовільняє заданій умові.

arrayName**.some (**(element, index, baseArray) => умова )

-------------- Визначити чи є і масиві числа менші 5 -------

 let arr = [-10, -9, 5, 13, 17, 16, 18, 19, 21, 11, 7, -4]

      if (arr.some((temp) => temp < 5)) document.write('Yes')

      else document.write('No')

Метод «**.sort**» Сортування (упорядкування)

«**.sort()**» Використовується для сортування **string.** Якщо вказуються порожні дужки, тоді масив буде упорядкований у порядку зростання, при цьому усі елементи будуть перетворені у **string**

let arr = [5, 2, 17, 19, 21, 11, 7]

arr.sort() // [ 11, 17, 19, 2, 21, 5, 7 ]

-------------- Упорядкувати масив чисел за зростанням -------

let names = ['Nona', 'John', 'Sara', 'Peter', 'Monica', 'Andrew', 'Niko']

      names.sort()

«**.sort(**el1, el2**)**» Виконується для сортування за вказаним правилом порівняння двох елементів.

-------------- Упорядкувати масив чисел за зростанням -------

 let arr = [-10, -9, 5, 13, 17, 16, 18, 19, 21, 11, 7, -4]

      arr.sort((el1, el2) => el1 - el2)

-------------- Упорядкувати масив чисел за спаданням -------

 let arr = [-10, -9, 5, 13, 17, 16, 18, 19, 21, 11, 7, -4]

      arr.sort((el1, el2) => el2 – el1)

-------------- Упорядкувати масив імен за зростанням за другою буквою -------

  let names = ['Nona', 'John', 'Sara', 'Peter', 'Monica', 'Andrew', 'Niko']

      names.sort((name1, name2) => {

        if (name1[1] > name2[1]) return 1

        else if (name1[1] < name2[1]) return -1

        else return 0

      })

Метод «**.reverse**» Обертання масиву

«**.reverse()**» Використовується для розміщення елементів у зворотньому напрямку

arrayName**.reverse()**

let arr = [-10, -9, 5, 13, 17]

arr.reverse() // [17, 13, 5, -9, -10]

Метод «**.concat**» Об'єднання масиву

«**.concat()**» Повертає новий масив, який складається з елементів двох масивів.

let a = [1, 2, 3]

let b = [4, 5]

let c = a.**concat**(b) // [1, 2, 3, 4, 5]

let d = a.**concat**(b, c) // [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]

let f = a.**concat**(8, 9, 10) // [1, 2, 3, 8, 9, 10]

Метод «**.join**» Створення рядка з елементів масиву

«**.join()**» Створює рядок, елементи якого розділяються вказаним знаком-розділювачем

arrayName**.join**(знак-розділювач)

let a = [Марія, Петро, Василь, Іван]

let b = a.**join**(**;**) // “Марія;Петро;Василь;Іван”

let c = a.**join**() // Порожні дужки – це кома “Марія,Петро,Василь,Іван”

Метод «**.split**» Створення масиву з рядка

«**.split()**» Створює масив, шляхом розбиття рядка, елементи якого розділяються вказаним знаком-розділювачем

рядок **.split** (знак-розділювач)

let a = “Марія;Петро;Василь;Іван”

let b = a.**split**(“,”) // [“Марія”,”Петро”,”Василь”,”Іван”] – масив з 4 елементами

let c = a.**split**() // [Марія;Петро;Василь;Іван] – масив з одним елементом

Функція **swap(**el1, el2**)**  Обмін значеннями між двома елементами

function swap(a, b) {

return [b, a]

}

let arr = [5,10]

= swap(x, y);

console.log("x:", x) // Виведе: x: 10

console.log("y:", y) // Виведе: y: 5

Деякі **методи сортування** (упорядковування) **масивів**

Порівняння алгоритмів сортування виконується за – часом виконання та кількістю затребуваної оперативної памяті.

При порівнянні двох алгоритмів, кращим завжди є той, який виконується за найменший період часу та споживає найменше оперативної памяті.

В залежності від (ситуації) умов використання алгоритму, доречним може бути використання алгоритму як з найдовшим часом виконання + мінімальним споживанням оперативної памяті, так і з з найшвидшим часом виконання + будь якою кількістю спожитої оперативної памяті.

Алгоритм **Сортування бульбашкою** (Один з Найгірших алгоритмів)

Виконує порівняння двох сусідніх елементів (списку чи масиву) .  
Якщо один з елементів, не відповідає критерію сортування (є більшим, або ж, навпаки, меншим за свого сусіда), то ці два елементи міняються місцями.   
Прохід по списку продовжуватиметься доти, доки дані не будуть відсортованими.  
Як тільки алгоритм виконає прохід, в результаті якого не буде змінено положення жодного елементу, це стає ознакою того що масив вже відсортований.

let changed

do{

changed = false

for(let i=**1**; i< a.length; i++) {

if( a[**i-1**] > a[i] ) {

let tmp = a[i-1];

a[i-1] = a[i]

a[i] = tmp

changed = true;

}

}

}

while(changed)

Алгоритм **Сортування змішуванням**

Це алгоритм сортування, який походить від бульбашкового сортування, але сортування відбувається в обох напрямках, міняючи напрямок при кожному проході.  
З початку виконується прохід по списку зліва на право (подібно до звичайного бульбашкового сортування), а потім зправа на ліво.  
У процесі проходження, більші елементи переставляються в кінець масиву, а менші - у початок.

let leftIndex = 0

let rightIndex = array.length - 1

while (leftIndex < rightIndex) {

for (let idx = leftIndex; idx < rightIndex; idx++) {

if ( array[idx] > array[idx + 1]) { swap(array, idx, idx + 1) }

}

rightIndex--;

for (let idx = rightIndex; idx > leftIndex; idx--) {

if ( array[idx] < array[idx - 1]) {swap(array, idx, idx - 1) }

}

leftIndex++;

}

Алгоритм **Сортування включенням** (вставок)

Сортування включенням полягає у тому, щоб послідовно переходити через масив та вставляти поточний елемент на відповідне місце у відсортованій частині масиву, яка знаходиться зліва від поточного елементу. На початку алгоритму відсортована частина масиву містить перший елемент, а потім кожен наступний елемент вставляється у правильне місце.

function insertionSort(arr) {

for (let i = 1; i < arr.length; i++) {

let currentElement = arr[i];

let j = i - 1;

while (j >= 0 && arr[j] > currentElement) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = currentElement;

}

return arr;

}

Алгоритм **швидкого сортування** (Quicksort)

Ідея алгоритму полягає в переставлянні елементів масиву таким чином, щоб його можна було розділити на дві частини і кожний елемент з першої частини був не більший за будь-який елемент з другої.

|  |  |
| --- | --- |
| function swap(A, i, j) {          let temp = A[i]          A[i] = A[j]          A[j] = temp  }  function Partition(A, p, q) {            let pivot = A[q] // базовий елемент             let i = p - 1            for (let j = p; j < q; j++) {                    if (A[j] <= pivot) {                        i = i + 1                      swap(A, i, j)                  }            }                  swap(A, i + 1, q)                  return i + 1 // повертаємо індекс pivot   } |  |
| function Quicksort(A, p, q) {          if (p >= q) return     // якщо індекс р >= індекса q, функція певертає результат          let i = Partition(A, p, q) // ділимо на дві частини  Quicksort(A, p, i - 1) // окремо сортуємо першу частину  Quicksort(A, i + 1, q) // окремо сортуємо другу частину  // Приклад Рекурсії – коли функція викликає сама себе  }        Quicksort(arr, 0, arr.length - 1) | |

**Двійковий** (бінарний) **пошук**

Двійковий (бінарний) пошук - алгоритм пошуку елемента у **впорядкованому** масиві.

Основна ідея двійкового пошуку полягає в тому, щоб порівняти шуканий елемент з елементом по середині масиву. Якщо елемент співпадає, пошук завершується і повертається індекс знайденого елемента. Якщо шуканий елемент менший за елемент по середині, пошук продовжується у лівій половині масиву, інакше - у правій половині.

Цей процес повторюється до тих пір, поки шуканий елемент не буде знайдений або область пошуку зменшиться до порожнього масиву. Якщо елемент не знайдений, алгоритм поверне спеціальне значення (наприклад, -1), що показує, що шуканий елемент відсутній у масиві.

function binarySearch(arr, searchEl) {

let left = 0;

let right = arr.length - 1;

while (left <= right) {

let mid = Math.floor((left + right) / 2)

if (arr[mid] === searchEl) {return mid} // Елемент знайдено, повертаємо його індекс або {return true} якщо число

if (arr[mid] < searchEl) {left = mid + 1} // Шуканий елемент у правій половині

if (arr[mid] > searchEl) {right = mid – 1} // Шуканий елемент у лівій половині

} return -1 // Індекс шуканого елемента не знайдено {return false} якщо числа

}

**Багатовимірні масиви**

Багатовимірний масив, це масив елементами якого є інші масиви (підмасиви) або об'єкти.

**Двовимірний масив** - кожен елемент двовимірного масиву є окремим одновимірним масивом.

let **arr** = [

[26,20,12],

[61,22,53],

[96,28,53]

]

console.log(arr[0][0]) // Виведе 26

console.log(arr[1][2]) // Виведе 53

console.log(arr[2][1]) // Виведе 28

Це приклад **двовимірного масиву** **arr**, що містить числові значення у вигляді (уявної) таблиці з трьома рядками та трьома стовпцями. Кожен рядок масиву є окремим одновимірним масивом, а всі вони утворюють двовимірний масив.

Для доступу до елемента двовимірного масиву використовуються два індекси. Перший індекс вказує на рядок, а другий - на стовпець, де знаходиться даний елемент.

Схематичне представлення двовимірного масиву в оперативній пам'яті.



Двовимірний масив з **різною кількістю елементів** у рядках

Масив з різною кількістю елементів у рядках, також відомий як "нерівномірний масив" або "рваний масив" (jagged array), є масивом, у якого кожен рядок може містити різну кількість елементів.

Задача. Знайти суму всіх елемнтів Нерівномірного масиву

|  |  |
| --- | --- |
| 0  1  2  3  4  5  let arr = [ [1, 2, 5, 2, 1], 0          [6, 2, 1, 9], 1        [1, 8, 5, 2, 2, 9], ] 2    let sum = 0  for (let i = 0; i < arr.length; i++) {        for (let j = 0; j < arr[**i**].length; j++) {        sum += arr[i][j]      }  } | 0  1  2  3  4  5  let arr = [ [1, 2, 5, 2, 1], 0          [6, 2, 1, 9], 1        [1, 8, 5, 2, 2, 9], ] 2    let sum = 0  for (const row of arr) {  for (const el of row) {  sum += el  }  } |

**Тривимірний масив**



Метод «**.flat**» Створення нового масиву з меншою вимірністю

**.flat** Призначений для створення нового масиву з меншою вимірністю елементів **depth** (за замовчуванням depth = 1)

назва\_масиву.**flat(depth)**

let arr = [[1, 2, 5, 2, [6, [2, 1], 9], [1, 8, [5, 2, 2], 9]] 1 ]

let arr2 = arr.**flat()** // теж саме що arr.**flat(1)** - видалить всередині 1 рівень дужок

// arr2 = [[1, 2, 5, 2, [6, 2, 1, 9], [1, 8, 5, 2, 2, 9]] 1 ]

let arr2 = arr.**flat(2)** // видалить всередині 2 рівня дужок

// arr2 = [[1, 2, 5, 2, 6, 2, 1, 9, 1, 8, 5, 2, 2, 9,] 1 ]

Якщо потрібно отримати одновимірний масив, незалежно від вкладеності елементів масиву, то можна викоритати ***Infinity***

let arr2 = arr.**flat(Infinity)** // видалить всередині всі дужки

// arr2 = [1, 2, 5, 2, 6, 2, 1, 9, 1, 8, 5, 2, 2, 9, 1]

Задача. Знайти максимальний елемент Нерівномірного масиву

       0  1  2  3  4  5

let arr = [ [1, 2, 5, 2, 1], 0

        [6, 2, 1, 9], 1

      [1, 8, 5, 2, 2, 9], ] 2

let maxNum = arr.flat(**Infinity**).reduce((prevmax, el) => (el > prevMax ? el : prevMax))

або

let maxNum = Math.max(…arr.flat(**Infinity**))

**JSON** **Копіювання** (клонування) багатовимірних **масивів та об’єктів**

Так копіювати багатовимірні масиви не можна!



**Універсальний спосіб створення копії багатовимірного масиву**

**JSON** (JavaScript Object Notation) - це легкий формат обміну даними, який використовується для передачі та збереження структурованих даних.

Копіювання двовимірних масивів з використанням функції **JSON.stringify**

Функція **JSON.stringify** допомагає здійснити перетворення різних типів даних JavaScript, такі як об'єкти, масиви, рядки, числа, булеві значення та значення null, у формат (рядок) JSON, що зручно для обміну даними з веб-серверами або іншими джерелами даних.

Важливою особливістю цієї функції є те, що вона автоматично перетворює значення, що не є допустимими в JSON, такі як функції або undefined, у значення null.

**JSON.parse** є вбудованою функцією в JavaScript, яка дозволяє перетворити рядок JSON (JavaScript Object Notation) на відповідне значення JavaScript, за наданою від JSON.stringify інструкцією.



**Вкладені функції –** функції, що викликаються в середині іншої функції

Вкладені функції (також відомі як внутрішні або локальні функції) є функціями, які оголошені і визначені всередині тіла іншої функції в програмуванні.   
**Важливо!** Якщо функція викликається в середині іншої функції, тоді для неї створюється ще один контекст виконання (Execution Context), який буде видалений, як тільки функція виконається.  
Це означає, що вкладені функції існують лише всередині зовнішньої функції і не можуть бути доступні або викликані ззовні цієї функції.   
Вкладена функція має доступ до змінних і параметрів (зовнішнього контекесту виконання) зовнішньої функції, включаючи ті, які передані їм в якості параметрів або аргументів, а також до параметрів глобального контексту виконання.

function outerFunction(x) {

function innerFunction(y) {

return y \* 2

}

let result = innerFunction(x)

return result

}

console.log(outerFunction(5)); // Виведе 10

**Рекурсія –** функція, що викликає сама себе, з іншими параметрами

Рекурсія - це техніка, при якій функція викликає саму себе з іншими параметрами.  
Такі звернення називаються рекурсивними викликами, а функція, що містить рекурсивні виклики, − рекурсивною.

\*\*\* Рекурсію використовують у ситуаціях, коли легко звести вихідну задачу до послідовності задач того ж виду, але з іншими вихідними даними.

Знаходження факторіала числа n

1. Рекурсивний алгоритм

function factorial(n) {

if (n === 0 || n === 1) {

return 1

} else {

return factorial(n - 1) \* n

}

}

КРАЩЕ УНИКАТИ ВИКОРИСТАННЯ РЕКУРСІЇ  
Тому що кількість рекурсивних викликів може бути будь якою, відповідно для кожного наступного рекурсивного виклику буде створюватись новий контекст виконання – таке явище дуже перевантажує оперативну память.

1. Нерекурсивний алгоритм знаходження факторіала

 function name(n) {

       let res = 1

       for (let i = 1; i <= n; i++) {

              res \*= i

      }

                return res

            }

**Лексичне середовище**

Кожна запущена функція, блок коду {...}, і скрипт в цілому мають внутрішній (прихований) асоційований об’єкт, відомий як *Лексичне середовище (Lexical Environment).*

Коли функція виконується, на початку виклику автоматично створюється внутрішнє лексичне середовище, в  
якому зберігається список локальних змін, та посилання на зовнішнє лексичне середовище.

Отже, є два лексичних середовища.



На початку кожного виклику makeCounter(), створюється новий об’єкт лексичного середовища для зберігання змінних конкретного виклику makeCounter.

Під час виконання функції makeCounter(), анонімна функція, яка складається з одного рядка: *return count++*, лише створються, не виконується.

Властивість **[[Environment]] -** зберігає посилання на лексичне середовище, в якому була створена функція.



Посилання на лексичне середовище в якому була створена анонімна функція, має вигляд {count: 0}.

Пізніше коли анонімна функція викликається знову, створюється нове лексичне середовище, а посилання на зовнішнє лексичне середовище для нього береться з [[Environment]]



Тепер, коли код всередині анонімної функції шукає змінну count, він спочатку шукає у власному лексичному середовищі (воно порожнє), потім у зовнішньому makeCounter(), де він її знаходить і змінює.



**Замикання** - функція ретурнить результат внутрішньої функції

!!! Кожного разу коли функція завершується (ретурнить результат), контекст виконання цієї функції видаляється.

Вар. 1) **Замикання виникає** коли функція first() ретурнить в якості результату АДРЕСУ внутрішньої функції second(), та зберігає цей результат (адресу) у змінній зовнішнього (або глобального) лексичного середовища.   
В результаті чого, контекст виконання функції first(), не видалиться, та буде існувати, до тих пір, поки в змінній з зовнішнього лексичного середовища зберігається адреса внутрішньої функції second().  
Така змінна вважається Замкнутою на функції second().

Є ще одне визначення, яке мені десь попалось.

Вар. 2) Замикання - Це властиівсть функції запам'ятовувати посилання на зовнішнє лексичне середовище, в якому вона була створена.

function createCounter(value) {

function increment() {

return value++

}

return increment // Повертаємо функцію increment, яка зберігає замикання навколо counter

}

let counter = createCounter(0) // Локальна змінна, яка буде закрита замиканням

document.write(counter()) // поверне 1

document.write(counter()) // поверне 2

--------------------------------------------------------------------------------------

 function getUrlGenerator(domain) {

         return function(url) {

                return `https://${url}.${domain}`

         }

 }

 let comUrl = getUrlGenerator('com')

 let uaUrl = getUrlGenerator('ua')

 document.write(`${comUrl('google')}<br>`)    // https://google.com

 document.write(`${comUrl('netflix')}<br>`)     // https://netflix.com

 document.write(`${uaUrl('google')}<br>`)       // https://google.ua

 document.write(`${uaUrl('netflix')}`)      // https://netflix.ua

**setTimeout**Таймер

**setTimeout** використовується для запуску виконання функції або виконання коду **Один раз** через певний інтервал часу, вказаний у мілісекундах.

***Встановлення таймеру***

let timeoutID = setTimeout(function, delay)

**function** - це функція або код, який потрібно виконати після закінчення вказаного інтервалу delay.

**delay** – затримка виконання функції (або коду) в мілісекундах.

function myFunction() {

console.log("Функція myFunction виконана!")

}

let timeoutID = setTimeout(myFunction, 2000) // Почекати 2 секунди перед виконанням функції myFunction

---------------------------------------------------------------------

let timeoutID = setTimeout( () => console.log("Функція myFunction виконана!"), 2000) // теж саме

***Передача додаткових параметрів у функцію***

let Змінна = setTimeout(function, delay, arg0, arg1)

**arg0, arg1** – параметри для функції

let invertalID = setTimeout(myFanc, 2000, ‘Hello’, ‘Ivan’)  
 function myFunction( greeting, name) {

console.log(`

${greeting} ${name}

`)

}

**clearTimeout** Зупинка таймера

**clearTimeout** використовується для скасування запланованого виконання функції, для цього функції .clearTimeout потрібно передати ідентифікатор таймера, який потрібно скасувати.

clearTimeout(timeoutId)

**timeoutId** - це ідентифікатор таймера, який повертається при виклику функції .setTimeout.

let intervalId = setTimeout( () => alert('Зареєструйтесь'), 5000)

if (confirm('Бажаєте зареєструватись?')) clearTimeout (intervalId)

Таймер **setInterval**

**setTimeout** використовується для запуску виконання функції або виконання коду **Багаторазово** через певний інтервал часу, вказаний у мілісекундах.

***Встановлення таймеру***

let invertalID = setInterval (function, delay)

**function** - це функція або код, який потрібно виконати після закінчення вказаного інтервалу delay.

**delay** – затримка виконання функції (або коду) в мілісекундах.

function myFunction() {

alert("Зареєеструйтесь!")

}

let interval ID = setInterval (myFunction, 2000) // кожні 2 секунди викличе myFunction

---------------------------------------------------------------------

let interval ID = setInterval ( () => alert("Зареєеструйтесь!"), 2000) // теж саме

***Передача додаткових параметрів у функцію***

let Змінна = setInterval (function, delay, arg0, arg1)

**arg0, arg1** – параметри для функції

let intervalID = setInterval (myFanc, 2000, ‘Hello’, ‘Ivan’)  
 function myFunction( greeting, name) {

console.log(`

${greeting} ${name}

`)

}

**clearInvertal** Зупинка таймера

**clearTimeout** використовується для скасування запланованого виконання функції, для цього функції .clearTimeout потрібно передати ідентифікатор таймера, який потрібно скасувати.

clearTimeout(timeoutId)

**timeoutId** - це ідентифікатор таймера, який повертається при виклику функції .setTimeout.

 let intervalId = setInterval( () => alert('Зареєструйтесь'), 3000)

 setTimeout(() => clearInterval(intervalId), 10000)

**Об’єкти (ООП)** Об'єктоорієнтоване програмування

Об'є́ктоорієнто́ване програмува́ння **ООП** – це є парадигма, тобто підхід до розробки програми, який полягає у попередньому аналізі задачі, створенню об’єктів (тобто сутностей, які ми описуємо за допомогою спеціальної програмної структури), та опису алгоритму взаємодії об’єктів між собою.

Основні поняття ООП в JavaScript:

1. **Клас** - це шаблон або модель, яка визначає структуру і поведінку об'єктів. В JavaScript класи можна створювати за допомогою ключового слова class.
2. **Об'єкт** - це екземпляр класу, який має властивості (поля) і методи (функції).
3. **Властивості** об'єкта - це дані, які зберігаються в об'єкті.
4. **Методи** об'єкта - це функції, які виконують дії або операції пов'язані з об'єктом.
5. **Наслідування** дозволяє створювати новий клас на основі існуючого класу, успадковуючи його властивості і методи. В JavaScript це можливо за допомогою ключового слова extends.
6. **Інкапсуляція** дозволяє обмежувати доступ до властивостей і методів об'єктів, щоб забезпечити більш чіткий і контрольований інтерфейс.
7. **Поліморфізм** дозволяє використовувати об'єкти різних класів з однаковим інтерфейсом, що дозволяє замінювати один об'єкт іншим без змін в коді.

**Асоціативний масив** (також відомий як **об'єкт** або словник) - це структура даних, що представляє собою набори пар «ключ»: «значення», або ж (що те ж саме) «властивість»: «значення» і дозволяють описувати об`єкти даних зберігаючи при цьому усі дані як єдину структуру.

**Ключ** (властивість) – це рядок (string) або символ (Symbol)

**Значення** – величина довільного типу

let об’єкт = {

ключ: значення,

}

Звернення до елементів об’єкта

let person = {

userName: "John",

age: 30,

city name: "New York"

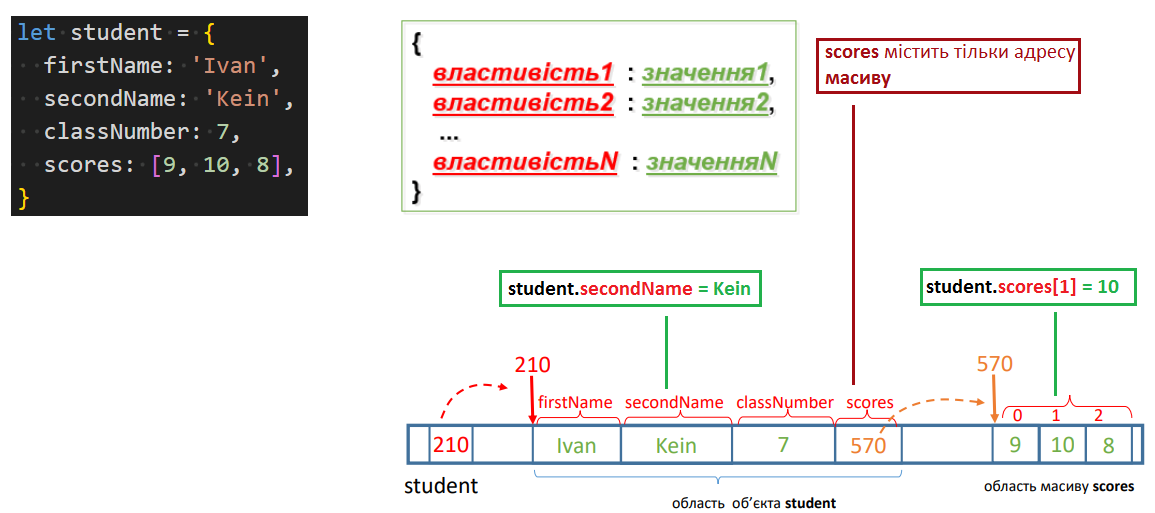
}

console.log(person**.**userName) // "John"

console.log(person**.**age) // 30

console.log(person[‘city name’]) // "New York"

Якщо ключ є недопустимим ідентифікаторам, наприклад, якщо містить пробіл або починається з цифри – тоді ключ записується так **student[‘first name’]**



let client = {

        title: {

          secondName: 'Smith',

          firstName: 'John',

          },

        adders: {

          zipCode: '88000',

          city: 'Uzhhorod',

          street: 'Svobody',

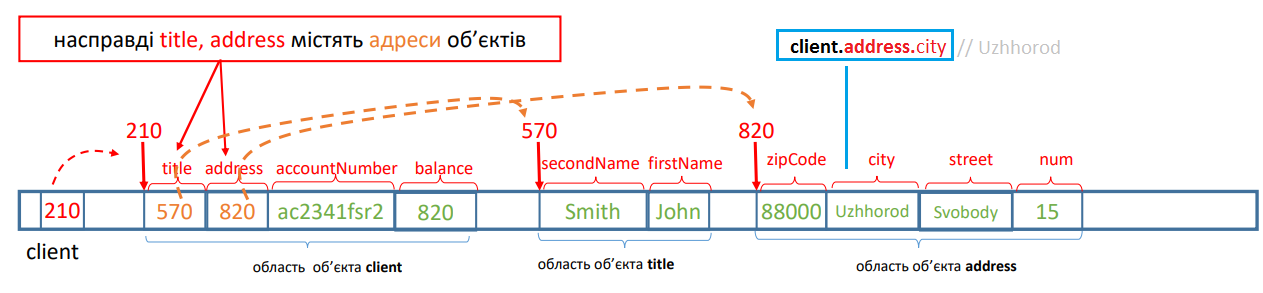
          num: 15,

          },

        accountNumber: 'ac2341fsr2',

        balance: 820,

      }



Динамічне **додавання** та **видалення** властивостей

Особливістю JavaScript є можливість динамічного додавання і видалення властивостей з об'єктів навіть після їх створення.

let person = {

name: "John",

age: 30

}

// **Динамічне додавання** новоих властивостей до об'єкта **person**

* якщо властість з такою назвою існує, то її значення буде змінено
* якщо властивості з такою назвою немає, то вона буде створена з вказаним значенням

person.name = “Tom”  
person.city = "New York"

person.job = "Engineer"

console.log(person) // Виведе: { name: "Tom", age: 30, city: "New York", job: "Engineer" }

// **Динамічне видалення** властивості з об'єкта, використовуємо оператор **delete**

delete person.job

console.log(person) // Виведе: { name: "Tom", age: 30, city: "New York" }

**Видалення** об'єкта

let person = {

name: "John",

age: 30

}

person = null // null – означає що адреса порожня

У цьому випадку змінна **person** буде мати значення **null**, а посилання на об'єкт { name: "John", age: 30 } буде втрачено. Якщо змінних які посилаються на цей об'єкт більше не має, він одразу ж видаляється з пам'яті за допомогою сбірника сміття (**garbage collector**).

« **in** » Перевірка існування властивості об’єкта

Перевірка існування властивості з певним іменем здійснюється за допомогою оператора « **in** »

let person = {

name: "John",

age: 30

}

alert(‘name’ **in** person) // true (тобто, така влиствість існує)

alert(‘secondName’ **in** person) // false (не існує)

« **for …** **in** » Перебір всіх властивостей **об’єкта**

Цикл **for...in** використовується для перебору всіх властивостей об'єкта, з виконанням певного коду для кожної властивості.

let price = {

firstValue: 300,

secondValue: 350,

threeValue: 400,

}

Синтаксис циклу **for...in** такий:

for (const key in price)

key – константа (або змінна), у яку поступово копіюються назви властивостей

price – ім'я об'єкта

**for** (const key **in** price) {

document.write(`key + ": " + (price[key] += 10)`)

}

Задача. Дано масив імен. Підхраувати кількість входжень кожного імені

const usersNames = ['Ivan', 'Petro', 'Olga', 'Ivan', 'Olga', 'Petro', 'Olga', 'Olga', 'Ivan']

let namesSum = {}

for (const name of usersNames) {

        if (name in namesSum) namesSum[name]++

        else namesSum[name] = 1

}

for (const key in namesSum) {

      document.write(`${key} - ${namesSum[key]}<br>`)

}

Обчислювані назви властивостей

Обчислювані назви властивостей (англ. computed property names) - це можливість створювати об'єкти з динамічними ключами (назвами властивостей) за допомогою спеціального синтаксису.

let position = “driver\_”

let id = 12

let dynamicAge = “age”

let person = {

name: “John”

[dynamicAge]: 30  
 [`${position}${id}`]: “school bus”

}

console.log(person) // Виведе: { name: "John", age: 30, driver\_12: "school bus" }

Створення об’єкта з змінних і констант

Коли назва властивості об'єкта співпадає з назвою змінної, з якої ми отримуємо значення, можна використовувати наступний скорочений запис

let model = “Audi”

let weight = 2100

let owner = “Ivan”

let auto = {

model,

weight,

owner,

}

console.log(auto.model) // Виведе: "Audi"

console.log(auto.owner) // Виведе: “Ivan”

«**Object.fromEntries()** » Створення об’єкта з масивів пар

Створення об'єкта з масивів, у вигляді пари – (ключ – значення)

Якщо маємо набір властивостей (ключів) і відповідних значень у формі масивів, з такого масиву можна створити об’єкт з використанням **Object.fromEntries()**

[ [ «ключ1», «значення1» ], [ «ключ2», «значення2» ], . . . , [ «ключN», «значенняN» ] ]

const keyValuePairs = [

["name", "John"],

["age", 30],

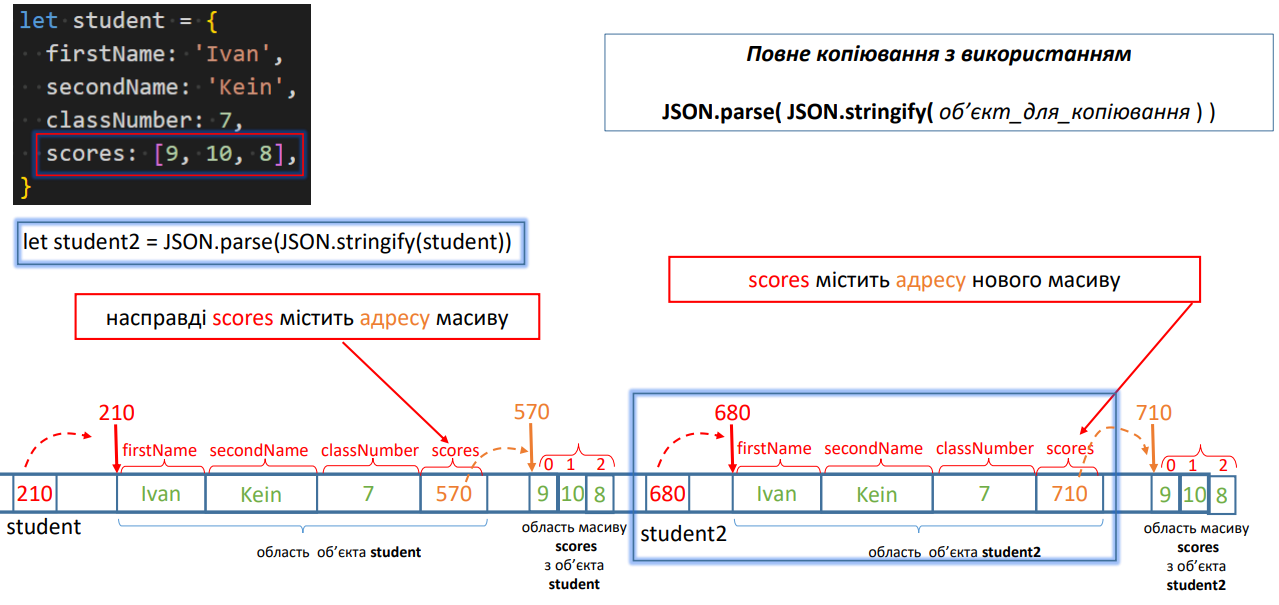
["occupation", "Engineer"]

]

const person = Object.fromEntries(keyValuePairs);

console.log(person) // Виведе: { name: "John", age: 30, occupation: "Engineer" }

**Копіювання** об'єктів



« **Object.assign** » **Копіювання** властивостей об'єктів

Копіювання властивостей з одного об’єкта в інший об’єкт з використанням **Object.assign**

Дозволяє об'єднати властивості декількох об'єктів в один об'єкт-ціль.

Синтаксис **Object.assign(target, ...sources)**:

* **target**: Цільовий об'єкт, до якого будуть скопійовані властивості. Цей об'єкт буде модифікований.
* **sources**: Один або декілька об'єктів, з яких будуть копійовані властивості у цільовий об'єкт. Ви можете передавати більше одного об'єкта через кому.

|  |  |
| --- | --- |
| let obj1 = {  model: “Audi”,  weight: 2100,  owner: “Ivan”,  }  Let obj2 = {  price: 35000,  fuel: “diesel”,  }  let obj3 = {  price: 50000  model: “BMW”  }  Object.assign(obj1, obj2, obj3)  odj1 = {  model: “BMW”  weight: 2100,  owner: “Ivan”,  price: 50000,  fuel: “diesel”,  } | Ще один варіант використання |

Якщо назви властивостей співпадають, тоді вони перезаписуються.

**Деструктуризація** об'єктів

Дозволяє присвоювати окремі властивості об'єкта до змінних, за допомогою спеціального синтаксису, або передавати окремі властивості об'єкта у функцію, в якості аргументів.

const person = {

name: "John",

age: 30,

occupation: "Engineer"

}

let { name, age } = person // Створить 2 змінні (name, та age) з значенням (“John”, та 30)

------------------------------------------

function getTotalPrice({price, productNumber}) {

return price \* productNumber

}

let objAuto = {

model: “Audi”,

weight: 2100,

price: 20000,

productNumber: 10

}

let totalPrice = getTotalPrice(objAuto)

**Методи** об'єктів

Метод – це функція яка описує деяку функціональну можливість об’єкта.

Об’єкти реальної дійсності характеризуються:

* **Властивостями-характеристиками** (деякими величинами (ім’я користувача, зріст, вік, ….))
* **Функціональними можливостями** (діями, що може виконати сам об'єкт, або можна виконати над об’єктом). Функціональні можливості об’єкта описують за допомогою методів (властивостей – функцій).

З**вертання до методів**

**об'єкт.метод**(аргументи)

**Об'єкт** - це змінна, що містить об'єкт.

**Метод** - назва методу, який викликаємо. Якщо метод приймає аргументи, вони передаються їх у дужках.

let **book** = {

title: 'JavaScript: The Good Parts',

author: 'Douglas Crockford',

year: 2008,

**getBookInfo**: function() {

return this.title + ' - ' + **this**.author + ' (' + **this**.year + ')'

}

}

let bookInfo = **book.getBookInfo()** // або так **-** let bookInfo = **book[‘getBookInfo’]()**

console.log(bookInfo) // Виведе "JavaScript: The Good Parts - Douglas Crockford (2008)"

**this** – звертання до інших властивостей та методів

**this** - вказує на поточний об'єкт або контекст (лексичне середовище), в якому викликається функція або метод.

Значення **this** залежить від того, як саме була викликана функція або метод, і змінюється в залежності від контексту виклику.

**this** – сховище яке містить спільні властивості, які доступні для всіх методів

----------------------------------------------------------------- Задача ----------------------------------------------------------------------

let product = {

      // ----------- Властивості - характеристики ---------

         title: 'Coffee',

         price: 430,

         count: 32,

        storePricePerDay: 3,

        //------------ Методи (функіональні можливості) ------------

getTotalPrice: function () { return this.price \* this.count },

getStorePriceForDays: function (**daysCount**) { return this.storePricePerDay \* daysCount },

------- Якщо функція **тільки змінює** значення внутрішніх змінних, **get** не використовуємо -----

reducePrice: function (**percent**) {this.price = this.price \* (1 - percent / 100) },

increasePrice: function (**percent**) {this.price \*= 1 + percent / 100},

toString: function () { return `${this.title} : ${this.price}` },

valueOf: function () { return this.getTotalPrice() },

}

document.write(`Вартість зберігання за 10 днів: ${ product.getStorePriceForDays(**10**)} <br>`)

document.write(`Загальна вартість: ${product.getTotalPrice()} <br>`)

product.increasePrice(**20)**; document.write(`Збільшили ціна на **20%**: ${product.getTotalPrice()} <br>`)

Спеціальні методи об’єктів **«toString»**  **«valueOf»**

**toString()** - метод, що використовується при перетворенні об’єкта до рядка.  
За замовчуванням, метод **toString()** повертає рядок, який містить репрезентацію

об'єкта у вигляді **[object Object]**

**valueOf() -** метод, що використовується для перетворення об'єктів на примітиви (наприклад, числа, рядки, булеві значення)

Приклад перевизначення методу **toString()** та **valueOf()**

let person = {

name: 'John',

age: 30,

toString() { return `Person: ${this.name}, Age: ${this.age}` },

valueOf() { return `New age: ${this.age}` },

}

document.write(**person**) // Автоматично виведе рядок "Person: John, Age: 30"

document.write(**person \* 2**) // Автоматично виведе число "New age: 60"

**toString()** та **valueOf()** дозволяють забезпечити контроль над перетворенням обёєктів у рядок та примітивині значення (числа, рядки, булеві значення).

**Object.freeze** заборона змінення властивостей об`єкта

**Object.freeze** - призначений для заморожування властивостей об'єкта першого рівня (внутрішнього лексичного середовища), тобто робить його не змінюваним.

Коли об'єкт заморожений, жодні нові властивості не можуть бути додані, а існуючі властивості не можуть бути змінені або видалені. Заморожування об'єкта створює іммутабельний об'єкт, тобто об'єкт, який не може змінюватись після свого створення.

const myObject = {

name: 'John',

age: 30,

}

Object.freeze(myObject) // Заморожуємо об'єкт

myObject.age = 35 // Спроба змінити властивість буде ігнорована

myObject.gender = 'male' // Спроба додати нову властивість буде ігнорована

delete myObject.name // Спроба видалити властивість буде ігнорована

console.log(myObject) // Виведе: { name: 'John', age: 30 }

**Object.keys**  повертає **назви** властивойстей об`єкта

**Object.keys() -** призначений для отримання масиву із властивостей об'єкта, де кожен елемент масиву представляє ім'я однієї властивості об'єкта.

const person = {

name: 'John',

age: 30,

gender: 'male',

}

const keys = Object.keys(person)

console.log(keys) // Виведе: ['name', 'age', 'gender']

**Object.values**  повертає **значення** властивойстей об`єкта

**Object.values()** - це метод, який дозволяє отримати масив значень усіх властивостей об'єкта. Він повертає масив, де кожен елемент масиву представляє значення однієї властивості об'єкта.

const person = {

name: 'John',

age: 30,

gender: 'male',

}

const values = Object.values(person)

console.log(values) // Виведе: ['John', 30, 'male']

**Функції - фабрики**

**Функції-фабрики** (або фабричні функції) - це функції, які створюють і повертають нові об'єкти з певними властивостями і методами. Зазвичай вони використовуються для створення об'єктів певного типу з певними характеристиками.

1. Чітке розділення створення об'єкта від його логіки.
2. Можливість створювати об'єкти з однаковими властивостями, але з різними значеннями.
3. Зручність під час створення складних об'єктів з багатьма параметрами.

 function createUser(name, age, email) {

return {

name: name,

age: age,

email: email,

sayHello: function() {

console.log(`Привіт, мене звати ${this.name} і мені ${this.age} років.`)

},

}

}

// Використання фабричної функції для створення об'єктів користувачів

const user1 = createUser('John', 30, 'john@example.com')

const user2 = createUser('Jane', 25, 'jane@example.com')

**Конструктори об’єктів**

**Конструктори об'єктів** - це функції, які використовуються для створення нових об'єктів з однаковою структурою і функціональністю.

Потрібно спочатку оголосити функцію, яку ми будемо використовувати як конструктор, а потім створити новий об'єкт з використанням оператора **new**. При використанні new, функція-конструктор автоматично створить новий об'єкт і встановить його як значення **this** всередині функції.

Якщо ми описуємо об'єкт реальної дійсності, прийнято назву функції-конструктора завжди починати з **В**еликої летери, та **без** приставки **get**

function User(name, age, email) {

this.name = name;

this.age = age;

this.email = email;

this.sayHello = function() {

console.log(`Привіт, мене звати ${this.name} і мені ${this.age} років.`)

}

}

// Використання конструктора для створення об'єктів користувачів

let user1 = new User('John', 30, 'john@example.com')

let user2 = new User('Jane', 25, 'jane@example.com')

// Використання конструктора для створення масиву з об'єктами користувачів

let team = [

       new User('John', 30, 'john@example.com'),

       new User('Jane', 25, 'jane@example.com'),

       new User('Mark', 22, 'Mark@example.com'),

       new User('Jac', 32, 'jac@example.com'),

]

      console.log(team)

**« prototype »** Прототип

**Прототип** - **це об'єкт**, сховище яке містить спільні властивості і методи, які доступні для всіх об'єктів, створених за допомогою однієї функції-конструктора. Кожен об'єкт функції-конструктора має посилання на свій прототип, і він успадковує всі властивості і методи прототипу.

function **Person**(name, age) { // **Функція-конструктор**

this.name = name

this.age = age

}

**Person**.**prototype**.**greet** = function() { // Додавання **методу** до **прототипу функції-конструктора**

console.log(`Привіт, мене звати ${this.name} і мені ${this.age} років.`)

}

const **person1** = new **Person**('Олег', 25) // Створення нового **об'єкта** з допомогою **конструктора**

const **person2** = new **Person**('Ірина', 30)

**person1**.**greet()** // Виведе: Привіт, мене звати Олег і мені 25 років.

**person2**.**greet()** // Виведе: Привіт, мене звати Ірина і мені 30 років.

Метод **greet()** визначений на прототипі **Person**, але може бути викликаний для будь-якого **об'єкта**, створеного за допомогою функції-конструктора **Person**. Це дозволяє поділитись одним і тим же методом між різними об'єктами, що забезпечує економію пам'яті і допомагає покращити ефективність коду.

**--------------------- Задача. Гра «Рулетка» ------** з використанням функції конструктора **------------**

----- Властивості -------

* кількість полів рулетки
* мінімальне значення балів
* максимальне значення балів
* список згенерованих значень

----- Методи ----------

* генерування полів рулетки
* виведення списку згенерованих значень
* приведення до рядка
* крутити рулетку (отримання випадкового балу)
* метод гри (користувач крутить рулетку поки не відмовиться)

Поки користувач крутить рулетку, визначаємо рандомне значення рулетки та додаємо до загальної суми балів. Повідомляємо користувача про результат та загальну кількість балів

**//---- Властивості (у кожного свої) -------**

function Roulette(cellsNumber, minScore = -100, maxScore = 100) {

        this.cellsNumber = cellsNumber // кількість клітинок

        this.minScore = minScore // мінімальне

        this.maxScore = maxScore // максимальне

        this.gameField = this.generateGameField() // генеруємо поле з балами

}

**// ----- Методи (спільні для всіх) ----------**

// -------- генерування рандомного числа -------

Roulette.prototype.getRandomNumber = function (minValue, maxValue) {

        minValue ??= this.minScore //якщо minValue===**undefined** або **Null** то minValue = this.minScore

        maxValue ??= this.maxScore

        return minValue + Math.floor(Math.random() \* (maxValue - minValue + 1))

}

      // -------- генерування полів рулетки з рандомними числами --------

Roulette.prototype.generateGameField = function () {

        let gameField = []

        for (let cellNum = 0; cellNum < this.cellsNumber; cellNum++) {

          let randScore = this.getRandomNumber()

          gameField.push(randScore)

  }

        return gameField

}

let r1 = new Roulette(20, -500, 500)

      // -------- виведення списку згенерованих значень --------

Roulette.prototype.showField = function () {

        document.write(this.gameField)

}

      // -------- приведення до рядка -----------

Roulette.prototype.toString = function () {

        return `Roulette: ${this.minScore} - ${this.maxScore}`

}

      // -------- крутити рулетку (отримання випадкового балу) --------

Roulette.prototype.getRandomScore = function () {

        const randomIndex = this.getRandomNumber(0, this.cellsNumber)

        return this.gameField[randomIndex]

}

       // -------- Гра | метод playGame, що ініціалізує запуск гри --------

Roulette.prototype.playGame = function () {

        let totalSum = 0

        while (confirm('Хочете ще грати?')) { // поки користувач хоче грати

          const randScore = this.getRandomScore() // визначаємо рандомне значення рулетки

          totalSum += randScore // додаємо до загальної суми

          alert(`Score =${randScore}/nTotal = ${totalSum}`) // повідомляємо користувача про результат та загальну кількість балів

        }

}

let roulette1 = **new** **Roulette**(10, -200, 200)

roulette1.playGame()

**« class »**

**class** - це шаблон для створення об'єктів (об'єктно-орієнтовані структури), які мають спільні властивості та методи.

// Задача. Розробити клас «Передбачувач». Через кожні вказані кількість секунд отримувати передбачення

class **Predictor {**

       //--- властвості-характеристики (у кожного свої, і можемо використати у інших методах):

**constructor**(predictionInterval, predictionList) {

this.predictionInterval = predictionInterval

this.predictionList = predictionList

}

        // ----- Методи:

        // -------- вибір випадкового передбачення --------

        getRandomPrediction() {

          const randIndex = Math.floor(Math.random() \* this.predictionList.length)

          return this.predictionList[randIndex]

        }

        // ------- метод **run**, що ініціює запуск таймера і генерування передбачень --------

        run() {

          setInterval(() => {

            console.log(this)

            alert(this.getRandomPrediction())

          }, this.predictionInterval)

        }

**}**

let pr1 = new Predictor(1000, ['PEACE', 'LOVE', 'MONEY', 'BIG MONEY', 'BEER', 'Шовдарь', 'Пікниця',])

// console.log(pr1)

pr1.run()

**Позичання (**додавання**)** функцій-методів

Взяття методу одного об'єкта і використання його для іншого об'єкта, незалежно від того, який клас або прототип відповідає за метод.

**об’єкт\_2** . **назва\_метода** = **об’єкт\_1** . **назва\_додаваємого\_метода**

let obj1 = {

        prop1: 11,

        prop2: 22,

        showProp1: function () {

            document.write(this.prop1)

        },

}

let obj2 = {

        prop1: 21,

        prop2: 23,

}

     obj2.myShow = obj1.**showProp1** //Копіюємо метод (функцію)

     obj2.myShow()     //  поверне 21

**call()** Виклик функції з наперед заданим this.

Під час виклику функції можна задати (змінити) контекст **this**

**функція**.**call**( новий\_контекст, аргумент\_1, аргумент\_2, …)

let obj1 = {

        prop1: 11,

        prop2: 22,

        showProp1: function () {document.write(this.prop1) },

getSum( **val1, val2** ) {return this.prop1 + val1 + val2},

}

let obj2 = {

        prop1: 21,

        prop2: 23,

}

document.write(obj1. **showProp1** . **call** ( **obj2** )) // поверне 21

let s = obj1. **getSum** . **call** ( **obj2**, **10, 7** ) // поверне 38

document.write(s)

**apply()** Виклик функції з наперед заданим this.

Під час виклику функції можна задати контекст (змінити) **this. apply** передає аргументи у вигляді масиву

**функція**.**applyl**( новий\_контекст, [aргумент\_1, аргумент\_2, …])

let obj1 = {

… … … …

obj1. showProp1 . apply ( obj2 ) // поверне 21

let s = obj1. getSum . apply ( **obj2**, **[** **10**, **7 ]** ) // поверне 38

document.write(s)

**bind** Фіксування контексту

Метод **bind()** використовується для "фіксування" контексту виконання функції. Контекст в даному випадку визначається через значення this всередині функції. Цей метод дозволяє встановити, яке значення this буде використовуватися всередині функції, незалежно від того, як вона була викликана.

const **person** = {

name: 'John',

sayHello: function() {

console.log(`Hello, ${this.name}!`)

}

}

const greet = person.sayHello // Копіюємо метод (функцію) - Втрачаємо контекст

const firstGreet = person.sayHello() // Викликаємо метод (функцію) - Контекст не втрачається

const secondGreet = person.sayHello.**bind**(**person**) // Копіюємо метод (функцію) - **Фіксуємо** контекст

greet() // Виведе "Hello, undefined!

firstGreet() // Виведе "Hello, John!"

secondGreet () // Виведе "Hello, John!"

**person**: Об'єкт, метод (функція) якго ми викликаємо

**person**: Об'єкт, контекст якого буде використовуватись при виклику методу (функції).

Отже, коли ви створюєте прив'язану функцію boundFunction та викликаєте її, контекст буде завжди тим, який ви вказали у bind().

**bind** Фіксування контекста і частковим заданням значення параметрів функцій

Основна ідея полягає в тому, щоб заздалегідь задати деякі аргументи функції, а потім при виклику прив'язаної функції передати тільки залишкові аргументи.

const calculator = {

baseValue: 10,

multiply(factor) {

return this.baseValue \* factor;

}

}

const double = calculator.multiply.bind(calculator, 2) // Фіксуємо об'єкт та перший аргумент

**call** та **bind** В чому різниця?

Метод **call** **викликає** функцію згідно вказаного контексту та переданим аргументам

const person = {

name: 'John',

greet: function(message) {

console.log(`${message}, ${this.name}!`)

}

}

const newPerson = { name: 'Alice' }

person.greet.**call**(newPerson, 'Hello') // Виведе "Hello, Alice!"

Метод **bind** **створює нову** функцію, фіксуючи вказаний контекст, та при необхідності, з частково заданими аргументами. Він не викликає функцію відразу, а повертає нову функцію з фіксованим контекстом та, при необхідності, аргументами.

const person = {

name: 'John',

greet: function(message) {

console.log(`${message}, ${this.name}!`)

}

}

const newPerson = { name: 'Alice' }

const boundGreet = person.greet.**bind**(newPerson)

boundGreet('Hello') // Виведе "Hello, Alice!"

Отже, в основному різниця між ними полягає в тому, що call викликає функцію відразу, змінюючи контекст, а bind створює нову функцію з фіксованим контекстом і (опціонально) аргументами, яку ви можете викликати пізніше.

------------------------------------------------------- Задача ----------------------------------------------------------

Дано функцію без формальних параметрів. Знайти суму усіх аргументів, які буде передано у функцію,

використатавши reduce стосовно argumnets.

      function getSum( ) { // функція без формальних параметрів

        return [ ].reduce.call(arguments, (prevSum, el) => prevSum + el)

        arguments.reduce((prevSum, el) => prevSum + el) //так не можна, бо arguments це не масив

      }

      let s = getSum(2, 1, 9, 87, 98)

      document.write(s)

**this** у **функціях**

* **Звичайні (не стрілкові)** функції, беруть контекст з лексичного середовища в момент Виклику   
  Під час **виклику** **звичайної функції** **this =window !!!**

1. Така змінна – « **var** myVar» - являється властивістю window (this)

function **testFunc()** {

        let myVar = 0

        console.log(this)       // window

        document.write(this.myVar)     // window.myVar = 22

}

**var** myVar = 22   //  window.myVar = 22

**testFunc()**

1. Така змінна - « **let** myVar» - не являється її як властивість window (this)

function **testFunc()** {

        let myVar = 0

        console.log(this)       // window

        document.write(this.myVar)     // undefined

}

**let** myVar = 22

**testFunc()**

**3)** При використанні «**use strict**» this = **undefined**

‘**use strict**’

function **testFunc()** {

        let myVar = 0

        console.log(this)       // undefined

        document.write(this.myVar)     // undefined.myVar – error

}

**var** myVar = 22

**testFunc()**

* **Стрілкові** функції, беруть контекст з лексичного середовища в момент Опису (і фіксують його)  
  Під час **виклику** **стрілкової функції** **this = obj1 !!!**

let obj1 = **{**

        myVar: 77,

        method1: function () {

            let func = () => {

                console.log(**this**)            // this = obj1

                document.write(**this**.myVar)   // obj1.myVar = 77

            }

        return func

        }

**}**

let f = obj1.method1()

**f()**

**Інкапсуляція**

**Інкапсуляція** є одним з основних принципів об'єктно-орієнтованого програмування (ООП) і відображає ідею приховання деталей реалізації об'єкта від зовнішнього світу, забезпечуючи доступ лише до необхідного інтерфейсу. Це дозволяє зберігати код більш організованим, зменшує залежності між різними частинами програми та сприяє підтримці і модифікації коду.

Основні аспекти інкапсуляції:

1. Захищений стан: Об'єкти мають внутрішній стан (змінні та дані), які можуть бути приховані від прямого доступу і змінюватися лише через публічні методи.
2. Публічний інтерфейс: Об'єкти повинні мати публічний інтерфейс (методи та властивості), через який зовнішній світ може взаємодіяти з ними. Цей інтерфейс визначає доступні операції.
3. Захист від недопустимих змін: Внутрішні деталі реалізації можуть бути приховані, щоб зменшити можливість некоректного використання або змін.
4. Контроль доступу: Інкапсуляція дозволяє контролювати доступ до внутрішніх даних і функцій об'єкта. Це допомагає уникнути недопустимих змін і забезпечує правильну поведінку об'єкта.

class BankAccount {

constructor(initialBalance) {

let balance = initialBalance

}

this.deposit = function(amount) {

if (amount > 0) {

balance += amount

}

}

this.withdraw = function(amount) {

if (amount > 0 && amount <= balance) {

balance -= amount

}

}

this.getBalance = function() {

return balance

}

}

const account = new BankAccount(1000)

account.deposit(500)

account.withdraw(200)

console.log(account.getBalance()) // Виведе 1300

У цьому прикладі внутрішній стан об'єкта BankAccount (баланс) захищений від прямого доступу, і зміни можна здійснити лише за допомогою методів deposit та withdraw. Таким чином, інкапсуляція дозволяє управляти доступом до даних та забезпечити контрольовану взаємодію з об'єктом.

**#**  Приватні поля в класах

**Приватне поле** в класах - дозволяє створювати приватну змінну, в яку можливо записати тільки те значення, яке відповідає певній умові.

Приватні поля оголошуються за допомогою префікса **#** перед ім'ям змінної або методу. Внесення значень до них виконується за допомогою **set**, повернення значень – за допомогою **get**

class Pupil {

**#age**

        constructor(initName, initAge, initClass, initScore) {

          this.name = initName

          this.Age = initAge

          this.Class = initClass

          this.score = initScore

        }

**get** Age() {

          return this.#age

        }

**set** Age(newValue) {

          if (newValue < 7) **throw new Error**('Значення некоректне (має бути більше за 7)')

         else this.#age = newValue

        }

Статичне поле **Singlton**  - створення лише одного екземпляра класу

**Singleton** (одиночка) - це паттерн (шаблон або правила) проектування в об'єктно-орієнтованому програмуванні, який дає можливість створення Лише Одного Екземпляра певного класу і надає глобальну точку доступу до цього екземпляра.

Головна ідея паттерну Singleton полягає в тому, що клас має приватний конструктор, який не дозволяє створювати нові екземпляри класу поза ним самим. Він надає статичний метод для отримання цього єдиного екземпляра. Якщо екземпляр вже існує, метод повертає існуючий екземпляр, інакше він створює новий екземпляр та повертає його.

class CompanyCar{

           static **#companyCarRef** // Створили Приватний **Singleton** = Null

           constructor(driverName, carBrand, carNumber){

              if (CompanyCar.#companyCarRef) return CompanyCar.#companyCarRef

               else

                        this.driverName = driverName

                        this.carBrand = carBrand

                        this.carNumber = carNumber

                        CompanyCar.#companyCarRef = this

           }

}

let firstDriver = new CompanyCar('Володимир','MAN','ВН1111ЕХ')

let secondDriver = new CompanyCar('Сергій','DAF','ВН2222ЕХ')

console.log(`Другий водій - ${secondDriver}`) // Поверне: Другий водій - Володимир, MAN, ВН1111ЕХ

let thirdDriver = new CompanyCar('Микола','Volvo','ВН3333ЕХ')

console.log(`Третій водій - ${thirdDriver}`) // Поверне: Третій водій - Володимир, MAN, ВН1111ЕХ

CompanyCar.**#companyCarRef** = Null - Якщо **Singelton** приватний-команда проігнорується, так як запис нового значення в приватне поле виконується тільки через **set,** а його навіть не має.  
CompanyCar.**companyCarRef** = Null - Якщо **Singelton** публічний - команда виконається.

**static** Статичні поля та методи

Статичні поля і методи - це властивість класів у об'єктно-орієнтованому програмуванні. Вони пов'язані з класом в цілому, а не з конкретними екземплярами класу.

**Статичні поля:**

1. Статичні поля належать класу як цілому, а не окремому екземпляру класу.
2. Використовуються для зберігання спільних для всіх екземплярів класу, даних або значень.
3. Використовуються для збереження глобальних чи загальних даних, що відносяться до класу.

**Статичні методи:**

1. Статичні методи також належать класу як цілому, а не конкретним об'єктам класу.
2. Вони використовуються для виконання завдань, пов'язаних з класом в цілому, які не потребують доступу до інстанцій класу.
3. Зазвичай вони використовуються для створення відліків, утиліт, допоміжних функцій тощо.

class Car {

**static** totalCars = 0 // Статичне поле. Зміна значення

constructor(make, model) {

this.make = make // Звичайне поле

this.model = model // Звичайне поле

Car.totalCars++ // Звернення до статичного поля. Зміна значення

}

getFullName() {

return `${this.make} ${this.model}`

}

**static** getTotalCars() { // Статичний метод

return Car.totalCars // В середині статичних методів можна використовувати тільки статичні поля!!!

}

}

const car1 = new Car('Toyota', 'Camry')

const car2 = new Car('Honda', 'Civic')

console.log(car1.getFullName()) // Виведе "Toyota Camry"

console.log(car2.getFullName()) // Виведе "Honda Civic"

console.log(Car.getTotalCars()) // Звернення до статичного методу. Зчитування значення  
console.log(Car.totalCars) // Звернення до статичного поля. Зчитування значення  
Car.totalCars = 10 // Звернення до статичного поля. Зміна значення

**#static** Статичні **Приватні** поля та методи (get / set)

class Employee {

**static** **#**totalEmployees = 0 // **Статичне** приватне поле

constructor(name) {

this.name = name;

Employee.#totalEmployees++ // Збільшуємо лічильник при створенні нового працівника

}

**static** #getTotalEmployees() { // **Статичний** приватний метод

return Employee.#totalEmployees

}

static displayTotalEmployees() { // Статичний публічний метод

const total = Employee.#getTotalEmployees()

console.log(`Total employees: ${total}`)

}

}

const employee1 = new Employee('John')

const employee2 = new Employee('Jane')

Employee.displayTotalEmployees() // Виведе "Total employees: 2"

**Композиція** та **агрегація**

Композиція та агрегація - це два поняття, що відображають взаємозв'язок між об'єктами в об'єктно-орієнтованому програмуванні. Давайте розглянемо різницю на прикладі.

**Композиція** - це жорстке включення одного об'єкта в інший. Один об'єкт є часткою іншого об'єкта і не може існувати незалежно від нього. Це вказує на тісний зв'язок між об'єктами, де один об'єкт є складовою частиною іншого.

**Приклад композиції**:

class Engine {

start() {

console.log("Двигун запущено")

}

}

class Car {

constructor() {

this.engine = **new Engine()** // Композиція: машина має двигун

}

startCar() {

this.engine.start()

console.log("Машина поїхала")

}

}

const myCar = new Car()

myCar.startCar() // Виведе: "Двигун запущено" та "Машина поїхала"

У цьому прикладі об'єкт Car має внутрішній об'єкт Engine, і цей двигун є невід'ємною частиною машини.

**Агрегація** - це більш слабке включення одного об'єкта в інший. Один об'єкт може існувати незалежно, навіть якщо інший об'єкт включає його. Це вказує на більш вільний зв'язок між об'єктами.

**Приклад агрегації**:

class Author {

constructor(name) {

this.name = name;

}

}

class Book {

constructor(title, author) {

this.title = title;

this.author = **author** // Агрегація: книга має автора, але автор може існувати окремо

}

}

const author1 = new Author("John Doe")

const book1 = new Book("Book Title", author1)

console.log(book1.author.name) // Виведе: "John Doe"

У цьому прикладі об'єкт Book має властивість author, яка посилається на об'єкт Author, але автор може існувати окремо від книги.

Отже, головна різниця полягає в тому, наскільки тісно зв'язані об'єкти між собою та наявність у них незалежного існування.

Структура браузерних об’єктів **DOM** **BOM** і **JS**

Браузерні об'єкти складаються з трьох основних компонентів: DOM (Document Object Model), BOM (Browser Object Model) і JS (JavaScript).

1. **DOM** (Document Object Model): DOM - це структура, яка представляє структуру сторінки веб-сайту або документа HTML/XML в браузері. DOM визначає ієрархію елементів, таких як елементи HTML, їхні атрибути та тексти, як окремі об'єкти, з якими можна взаємодіяти за допомогою JavaScript. DOM включає в себе такі об'єкти, як document, element, node, attribute і т.д. DOM дозволяє змінювати структуру та вміст сторінки динамічно.
2. **BOM** (Browser Object Model): BOM - це набір об'єктів, які представляють браузер та його можливості, що виходять за межі об'єктів DOM. BOM включає в себе об'єкти, які дозволяють контролювати вікно браузера, навігацію, роботу з історією переходів, роботу з кукісами (cookies) та інші браузерні можливості. Деякі з популярних об'єктів BOM: window, navigator, location, history, screen та інші.
3. **JS** (JavaScript): JavaScript - це мова програмування, яка використовується для створення динамічної взаємодії на веб-сторінках. Вона дозволяє взаємодіяти з DOM та BOM, змінювати їх структуру, вигляд і поведінку. Це включає в себе додавання подій, зміну стилів, створення нових елементів, обробку даних з сервера та багато іншого.

**BOM** (Browser Object Model)

**BOM** (Browser Object Model) – об’єктна модель браузера, яка надає доступ до різноманітних функцій та об'єктів - управління вкладками, робота з історією перегляду, маніпулювання URL-адресами, взаємодія з користувачем через діалогові вікна, керування кукісами та іншими аспектами, які не пов'язані безпосередньо з DOM.

BOM включає в себе **window**, а також інші функції та об'єкти, що допомагають управляти браузерним середовищем та взаємодіяти з ним.

BOM об'єкт **window**

Об'єкт **window** є одним з основних об'єктів у браузерному середовищі і представляє вікно браузера. **window** є частиною Browser Object Model (BOM) і використовується для керування багатьма аспектами браузерного середовища та надає доступ до багатьох браузерних можливостей та функцій, пов'язаних з вікном та його оточенням.

**\*\*** **Window** є глобальним об’єктом JavaScript, через який програмними засобами можна звернутися до всіх інших об’єктів.

**Властивості window:**

location - посилання на об'єкт, що містить інформацію про поточну URL-адресу.

document - об'єкт document, який представляє DOM поточної сторінки.

history – посилання на об'єкт для керування історією переходів.

navigator - посилання на об'єкт, що надає доступ до властивостей та можливостей браузера.

screen - повертає посилання на об'єкт, що містить інформацію про властивості екрана.

frames – повертає масив елементів <iframe> поточного вікна.

opener – задає або отримує посилання на вікно, через яке було відкрито поточне вікно.

parent – повертає батьківське вікно, поточного вікна

self – повертає посилання на поточне вікно, або фрейм

status – повертає / встановлює текст у рядку стану у нижній частині браузера

**Властивості window. Розміри вікна:**

innerHeight - Висота внутрішньої області вікна браузера.

innerWidth - Ширина внутрішньої області вікна браузера.

outerHeight - Загальна висота вікна браузера, включаючи рамки та панель інструментів.

outerWidth - Загальна ширина вікна браузера, включаючи рамки та панель інструментів.

**Властивості window. Прокуртка вікна:**

pageXOffset (або scrollX) - Кількість пікселів, на яку сторінка була прокручена по горизонталі.

pageYOffset (або scrollY) - Кількість пікселів, на яку сторінка була прокручена по вертикалі.

screen - Об'єкт, що містить інформацію про властивості екрана, такі як розміри та глибина кольору.

screenX (або screenLeft) - горизонтальна позиція верхнього лівого кута вікна відносно екрана.

screenY (або screenTop) - вертикальна позиція верхнього лівого кута вікна відносно екрана.

**Методи window. Діалогові вікна**:

alert() - показує діалогове вікно з повідомленням.

confirm() - показує діалогове вікно з питанням "ОК" або "Скасувати".

prompt() – відображає діалогове вікно, з повідомленням і полем для вводу рядка.

**Методи window. Інтервали:**

setTimeout(func, delay) - виконує функцію після вказаного інтервалу часу.

setInterval(func, interval) - виконує функцію регулярно з інтервалом часу.

clearTimeout(timeoutID) - скасовує виконання функції, запланованого з setTimeout.

clearInterval(intervalID) - скасовує виконання функції, запланованого з setInterval.

**Методи window. Маніпуляції з вікнами:**

open(url, target, features) – Створює і відкриває нове дочірнє вікно або вкладку браузера.

close() – Закриває вікно яке було створено з використанням метода open()

focus() – Встановлює фокус на поточному вікні, щоб користувач міг взаємодіяти з ним.

blur() - Зняття фокусу з поточного вікна.

print() - Відкриття діалогового вікна для друку (печаті) вмісту поточної сторінки.

stop() – Зупиняє завантаження вікна

moveBy(x, y) – **Зміщує** вікно на вказану відстань (рх) від його поточного положення

moveTo(x, y) – **Переміщує** вікно згідно вказаних координат на екрані.

resizeBy(width, height) – збільшення / зменшення розмірів, на певне значення від поточних розмірів.

resizeTo(width, height) - Встановлює конкретні розміри для вікна, згідно width і height.

scrollBy(x, y) - прокручує вміст на вказану відстань від поточної позиції.

scrollTo(x, y) - встановлює конкретну позицію прокрутки.

postMessage(data, targetOrigin) - Відправка повідомлення іншому вікну або вкладці, відкритому за допомогою window.open(), за допомогою механізму Cross-Origin Communication.

addEventListener(event, handler) - Додає обробник події на вікно.

removeEventListener(event, handler) - Видаляє обробник події з вікна.

BOM об'єкт **location**

**Властивості та меотди об`єкта window.location:**

location.href - Повний URL-адрес поточної сторінки.

location.protocol - Протокол (наприклад, "http:", "https:") поточної сторінки.

location.host - Хост (доменне ім'я та порт) поточної сторінки.

location.hostname - Доменне ім'я (без порту) поточної сторінки.

location.port - Порт поточної сторінки.

location.pathname - Шлях (частина URL після доменного імені та порту) поточної сторінки.

location.search - Рядок запиту (частина URL після символу запиту ?) поточної сторінки.

location.hash - Частина URL після символу #, яка використовується для якорних посилань.

location.origin - Початкове походження (протокол, доменне ім'я та порт) поточної сторінки.

location.assign(url) - ініціює зміну поточної сторінки на іншу, вказану в параметрі url

BOM об'єкт **navigator**

**Властивості об`єкта window.navigator:**

navigator.userAgent - Рядок, який містить інформацію про браузер та операційну систему, який використовується для ідентифікації браузера на веб-сайтах.

navigator.platform - Рядок, який вказує на операційну систему, на якій запущено браузер (наприклад, "Win32", "MacIntel" тощо).

navigator.language - Рядок, який представляє мову, налаштовану на браузері користувача.

navigator.cookieEnabled - Булеве значення, яке показує, чи дозволені файли cookie в браузері.

navigator.onLine - Булеве значення, яке показує, чи пристрій підключений до мережі Інтернет.

navigator.plugins - Масив об'єктів, які представляють встановлені плагіни в браузері.

navigator.geolocation - Об'єкт, що надає доступ до геолокаційних можливостей пристрою.

navigator.mediaDevices - Об'єкт, який надає доступ до медіа-пристроїв, таких як веб-камера та мікрофон.

navigator.userAgentData - Об'єкт, який надає деталізовану інформацію про userAgent, зокрема можливості браузера та платформи.

BOM об'єкт **screen**

**Властивості об`єкта window.screen:**

screen.width - Ширина екрана в пікселях.

screen.height - Висота екрана в пікселях.

screen.availWidth - Доступна ширина екрана в пікселях (без врахування панелей інструментів).

screen.availHeight - Доступна висота екрана в пікселях (без врахування панелей інструментів).

screen.colorDepth - Глибина кольору екрана в бітах (кількість кольорів, яку екран може відображати).

screen.pixelDepth - Те ж саме, що і screen.colorDepth, але зазвичай використовується для більш точного вимірювання.

BOM об'єкт **history**

**Методи об`єкта window.history:**

history.back() - Перехід до попередньої сторінки в історії перегляду.

history.forward() - Перехід до наступної сторінки в історії перегляду (якщо була виконана дія history.back()).

history.go(n) - Перехід до сторінки в історії перегляду, розташованої на n позицій від поточної. Наприклад, history.go(-2) перенесе на дві сторінки назад.

history.pushState(state, title, url) - Додає новий запис до історії перегляду і змінює URL без перезавантаження сторінки.

history.replaceState(state, title, url) - Замінює поточний запис в історії перегляду і змінює URL без перезавантаження сторінки.

BOM об'єкт **- DOM** (Document Object Model)

**DOM** (Document Object Model) – об’єктна модель (ієрархічна структура) html документа

**Браузер:**

1. Отримав html документ
2. **Побудував DOM - об’єктну структуру html документа у памяті**
3. За допомогою **Rendering Engine** відобразив html документ (сайт)

Об'єкт **document** є однією з основних складових браузерних об'єктів і представляє собою DOM сторінки або документа, який відображається в браузері. Цей об'єкт надає доступ до структури та вмісту сторінки, і дозволяє здійснювати маніпуляції з елементами на сторінці за допомогою JavaScript.

Згідно DOM-моделі (Document Object Model), документ є ієрархією, деревом. **Кожен HTML-тег утворює вузол** дерева з типом «елемент». Вкладені в нього теги стають дочірніми вузлами. Для представлення тексту створюються вузли з типом«текст».

DOM (Document Object Model) включає **12 типів вузлів**, які представляють різні елементи структури документа HTML або XML:

1. Element Node (вузол елементу): Представляє HTML або XML елементи на сторінці. Наприклад, <div>, <p>, <a> і так далі.
2. Text Node (вузол тексту): Містить текстовий вміст в межах елемента. Наприклад, текст в середині <p> або span.
3. Attribute Node (вузол атрибуту): Представляє атрибути елементів, такі як href, src, class тощо.
4. Comment Node (вузол коментаря): Містить коментарі, які можна додавати в HTML або XML код.
5. Document Node (вузол документу): Представляє всю структуру документа. Це кореневий вузол.
6. DocumentType Node (вузол типу документа): Представляє тип документа, наприклад, <!DOCTYPE html>.
7. DocumentFragment Node (вузол фрагмента документа): Використовується для створення об'єктів, які є підмножиною документа, не прив'язаною до основної структури.
8. ProcessingInstruction Node (вузол обробки інструкції): Представляє обробні інструкції XML або інші типи документів.
9. CDATASection Node (вузол CDATA-секції): Містить неінтерпретований текст, такий як JavaScript або XML код.
10. EntityReference Node (вузол посилання на сутність): Використовується для заміни символів HTML або XML сутністю.
11. Entity Node (вузол сутності): Представляє символи, які можуть бути використані для заміни інших символів.
12. Notation Node (вузол нотації): Використовується для визначення нотацій в документі.

Ці типи вузлів утворюють ієрархію, яка представляє структуру документа і дозволяє програмно взаємодіяти з різними частинами сторінки.

Методи об’єкта document дял **вибірки елементів** (знаходження)

getElementById(id) - повертає елемент за його унікальним ідентифікатором (id).

getElementsByClassName(className) - повертає всі елементи, які мають певний клас.

getElementsByTagName(tagName) - повертає всі елементи, наприклад, всі <p> або <div>.

getElementsByName(name) - повертає всі елементи на сторінці, з вказаним **атрибутом** name.

querySelector(selector) - Вибирає перший елемент, який відповідає CSS-селектору selector.

querySelectorAll(selector) - Вибирає всі елементи, які відповідають CSS-селектору selector.

childNodes - Властивість, яка надає список дочірніх вузлів для елемента.

parentNode - Властивість, яка надає посилання на батьківський вузол для елемента.

Методи об’єкта documentдля **створення вузлів**

Створення вузлів в DOM (Document Object Model) дозволяє динамічно створювати нові елементи або текстові вузли, після чого додавати їх до структури веб-сторінки.

createElement(tagName) - створення нового HTML-елемента з вказаним ім'ям тегу.

createTextNode(text) - створення нового текстового вузла з вказаним текстовим вмістом

elem.cloneNode(true) - використовується для створення глибокої копії вказаного елемента DOM разом з усіма його дочірніми елементами і їхнім вмістом.

elem.cloneNode(false) - використовується для створення поверхневої копії.

Методи об’єкта documentдля **додавання вузлів**

parentElement.append() – додає елементи або рядки в кінець списку дочірніх елементівC

parentElement.prepend() – додає елементи або рядки на початок списку дочірніх елементів

element.before() – додає елементи або рядки перед елементом element

element.after() – додає елементи або рядки після елемента element

element.replaceWith() – змінює вузол element на заданий список вузлів або рядків

--------------------- Приклад --------------------------

const baseElement = document.getElementById('First')       // знаходимо базовий елемент орієнтир

const inp = document.createElement('input') // створюємо елемент

baseElement.after(inp)       // вставляємо створений елемент

Методи об’єкта documentдля **видалення вузлів**

remove() – можна викликати безпосередньо на елементі для його видалення.

let elementToRemove = document.getElementById("element")

elementToRemove.remove()

Властивості об’єкта documentдля **створення вмісту елемента**

innerHTML - дозволяє отримати або замінити (створити) HTML – вміст елемента у вигляді рядка

<div id="myDiv">Це елемент <strong>div</strong></div>

var myElement = document.getElementById("myDiv")

var content = myElement.innerHTML // Отримання вмісту

console.log(content) // Виведе Це елемент <strong>div</strong>

myElement.innerHTML = "Новий вміст <em>для</em> елемента" // Зміна вмісту

outerHTML – дозволяє отримати або замінити (створити) HTML – вміст елемента (включаючи сам елемент) у вигляді рядка

<div id="myDiv">Це елемент div</div>

var myElement = document.getElementById("myDiv") // Отримання вмісту

console.log(myElement.outerHTML) // Виведе <div id="myDiv">Це елемент div</div>

myElement.outerHTML = "<span id='newSpan'>Це новий span</span>" // Зміна вмісту

innerText - дозволяє отримати і встановити текстовий вміст елемента, ігноруючи будь-які HTML-теги, що можуть міститися всередині елемента.

<div id="myDiv">Це елемент <strong>div</strong></div>

var myElement = document.getElementById("myDiv")

var textContent = myElement.innerText // Отримання вмісту

console.log(textContent) // Виведе: Це елемент div

myElement.innerText = "Новий текст для елемента" // Зміна текстового вмісту

Властивості об’єкта document для **зміни стилів** **елементів**

Властивість **style:**

var myElement = document.getElementById("myElement")

myElement.style.color = "blue"

myElement.style.fontSize = "16px" // Тільки через **camelCase**

Метод **setAttribute()**:

var myElement = document.getElementById("myElement")

myElement.setAttribute("style", "color: blue; font-size: 16px;")

Властивості об’єкта document для **маніпуляцій з класами елементів**

className - властивість, яка використовується для доступу до атрибута class елемента HTML. Атрибут class дозволяє задавати один або кілька класів для елемента.

var myElement = document.getElementById("myElementId") // Отримання ID елемента

myElement.className = "newClass" // Заміна (присвоєння) нового класу

myElement.className += "additionalClass" // Додавання ще одного класу

classList - сучасна властивість, яка надає більш гнучкий і безпечний спосіб роботи з класами елементів, вона автоматично обробляє унікальність класів та допомогає уникнути дублювання імен.

var myElement = document.getElementById("myElementId") // Отримання ID елемента

myElement.classList.**add**("newClass") // **Додавання класу до списку класів**

myElement.classList.**remove**("classToRemove") // **Видалення класу з списку класів**

if (myElement.classList.**contains**("someClass")) { // **Перевірка наявності класу**

// ...

}

myElement.classList.**item**(0) // **Використання методу item для отримання класу за індексом**

myElement.classList.**length** // **Отримання кількості класів**

myElement.classList.**toggle**("toggleClass") // **Додавання або видалення класу в залежності від наявності**

Метод **toggle** властивості classList використовується для додавання класу до елемента, якщо його немає, або видалення класу, якщо він вже присутній. Це дозволяє змінювати стан класу елемента при кожному виклику методу toggle.

var myElement = document.getElementById("myElementId")

myElement.classList.toggle("active") // Перший виклик - додає клас - "active" доданий

myElement.classList.toggle("active") // Другий виклик - видаляє клас, оскільки він вже присутній

Цей метод корисний, коли потрібно створити простий механізм включення та виключення певного стану або стилів для елемента, наприклад, коли активувати/деактивувати вікно, меню або панель.

Методи об’єкта document **Атрибути**

setAttribute(‘name’, ‘value’) - Встановлює значення атрибуту

getAttribute(‘name’) - Читаємо значення атрибуту

removeAttribute(‘name’) - Видаляємо атрибут

hasAttribute(‘name’) - Перевіряємо чи існує атрибут

**Навігація за елементами**

const parentElement = childElement.parentElement - Найближчий батьківський елемент для childElement

const previousSibling = targetElement.previousElementSibling - Попередній сусідній елемент

const nextSibling = targetElement.nextElementSibling - Наступний сусідній елемент

const childElements = parentElement.children - Отримання всіх дочірніх елементів

const firstChild = parentElement.firstElementChild - Отримання першого дочірнього елементів

const lastChild = parentElement.lastElementChild - Отримання останнього дочірнього елементів

const parentElement = childElement.closest('.parent') - Знаходження найближчого батьківського елемента з класом "parent"

**Події**

**Події** (або події DOM) - це спосіб взаємодії з веб-сторінками через JavaScript. Вони виникають при різних діях користувачів чи програми, таких як клік мишею, натискання клавіші, завантаження сторінки та інші.

**Події документу (Document events)** - це події, які виникають на рівні всього документа

1. DOMContentLoaded Завантаження сторінки (весь HTML та структура DOM)
2. load Завантаження документа (весь контент сторінки, включаючи зображення та інші ресурси)
3. resize Користувач змінив розмір вікна (Frame, window)
4. beforeunload Виникає перед закриттям і дозволяє запитати користувача та визначити повідомлення для підтвердження.
5. unload Виникає після закриття сторінки, коли більшість ресурсів уже недоступні.
6. error Виникла помилка під час завантаження (Image, window)
7. move Користувач перемістив вікно (Frame, window)

**Події миші** - виникають під час взаємодії користувача з мишею на веб-сторінці.

1. сlick виникає, коли користувач клікає на елемент за допомогою лівої кнопки миші або виконує аналогічну дію на сенсорних пристроях.
2. mousedown виникає, коли користувач натискає ліву кнопку миші на елементі.
3. mouseup виникає, коли користувач відпускає кнопку миші після натискання на елементі.
4. dblclick виникає, коли користувач швидко подвійно клікає на елементі за допомогою лівої кнопки миші, тобто виконує подвійний клік.
5. mousemove виникає, коли користувач переміщує мишу над елементом на веб-сторінці.
6. mouseout виникає, коли миша покидає область елемента на веб-сторінці.
7. mouseover виникає, коли миша наводиться на елемент на веб-сторінці.
8. contextmenu це подія DOM, яка виникає, коли користувач намагається відкрити контекстне меню (контекстне вікно) на елементі, зазвичай правою кнопкою миші.
9. drag виникає, коли користувач починає перетягування об'єкта (натискаючи на нього лівою кнопкою миші або торкаючись на сенсорних пристроях) і розпочинає рух миші або пальця.
10. drop Ця подія виникає, коли користувач завершує перетягування та відпускає об'єкт над цільовою областю для розміщення (цільовий елемент)

**Події клавіатури**

1. keydown виникає в момент натискання клавіши на клавіатурі.
2. keyup виникає в момент відпускання клавіши, яка раніше була натиснута.
3. keypress виникає між подіями keydown та keyup

**Події елемента** - виникають під час взаємодії користувача або програми з конкретним елементом HTML

1. abort виникає, коли завантаження ресурсу (наприклад, зображення, аудіо, відео) було ненавмисно перервано або припинено, наприклад, через відміну завантаження або втрату з'єднання.
2. blur виникає, коли елемент втрачає фокус, тобто коли користувач перестає взаємодіяти з цим елементом, який був вибраний.
3. change виникає, коли значення елемента змінюється, зазвичай в результаті взаємодії користувача з елементами, які мають стани для вибору, такі як тексти, флажки або список випадаючого меню.
4. focus виникає, коли елемент отримує фокус, тобто коли користувач вибирає цей елемент для взаємодії, наприклад, для введення тексту у текстовому полі.
5. select виникає, коли користувач вибирає текст або змінює виділену область в елементах, які містять текст, таких як текстові поля або текстові області.
6. input виникає, коли користувач починає вводити дані в інпут

Обробка подій **- on**

**Обробка подій** - це процес виконання деякої команди, або функції, у відповідь на настання певних подій.  
**Задання обробника:**

* з використанням атрибуту елемента HTML.

У цьому способі необхідно HTML елементу додати атрибут, який починається з “on” і містить назву події (наприклад: onclick, onchange, onselect,…). Значенням атрибуту може бути або деяка команда, або виклик функції. Значення атрибутів вказується у лапках.

<body>

<button onclick="myFunction()">Натисніть мене</button>

<script>

function myFunction() {

alert("Кнопку натиснуто!")

}

</script>

</body>

* з використанням властивості елемента.

У цьому способі необхідно використати властивість елемента як об’єкта JS. Ім’я властивості має починатись з “on”подія.

<body>

<button id="myButton">Натисніть мене</button>

<script>

document.getElementById("myButton").onclick = function() {

alert("Кнопку натиснуто!")

}

</script>

</body>

Обробка подій **- addEventListener** та **removeEventListener**

Методи **addEventListener** та **removeEventListener** використовуються для додавання та видалення обробників подій на елементах DOM. Ці методи дозволяють керувати тим, як елементи реагують на різні події, такі як натискання кнопок миші, клавіш клавіатури, кліків тощо.

**addEventListener -** додає один або більше обробників подій на вказаний елемент.

**removeEventListener** - видаляє обробники подій з елементів. Для видалення обробника потрібно передати ті самі аргументи, які були використані при встановленні обробника

element.**addEventListener**(event, function, useCapture)

* **element**: Елемент DOM, на якому ви хочете встановити обробник події.
* **event**: Назва події, на яку встановлюється обробник (наприклад, 'click', 'mouseover', 'keydown' і т.д.).
* **function**: Функція, яка буде викликана при виникненні вказаної події.
* **useCapture** (необов'язково): Булеве значення, яке вказує, чи використовувати **режим захоплення** подій (true) чи **режим здійснення** подій (false).

------------------- Приклад ------------------------

const button = document.getElementById('myButton')

function handleClick() {

console.log('Button clicked!')

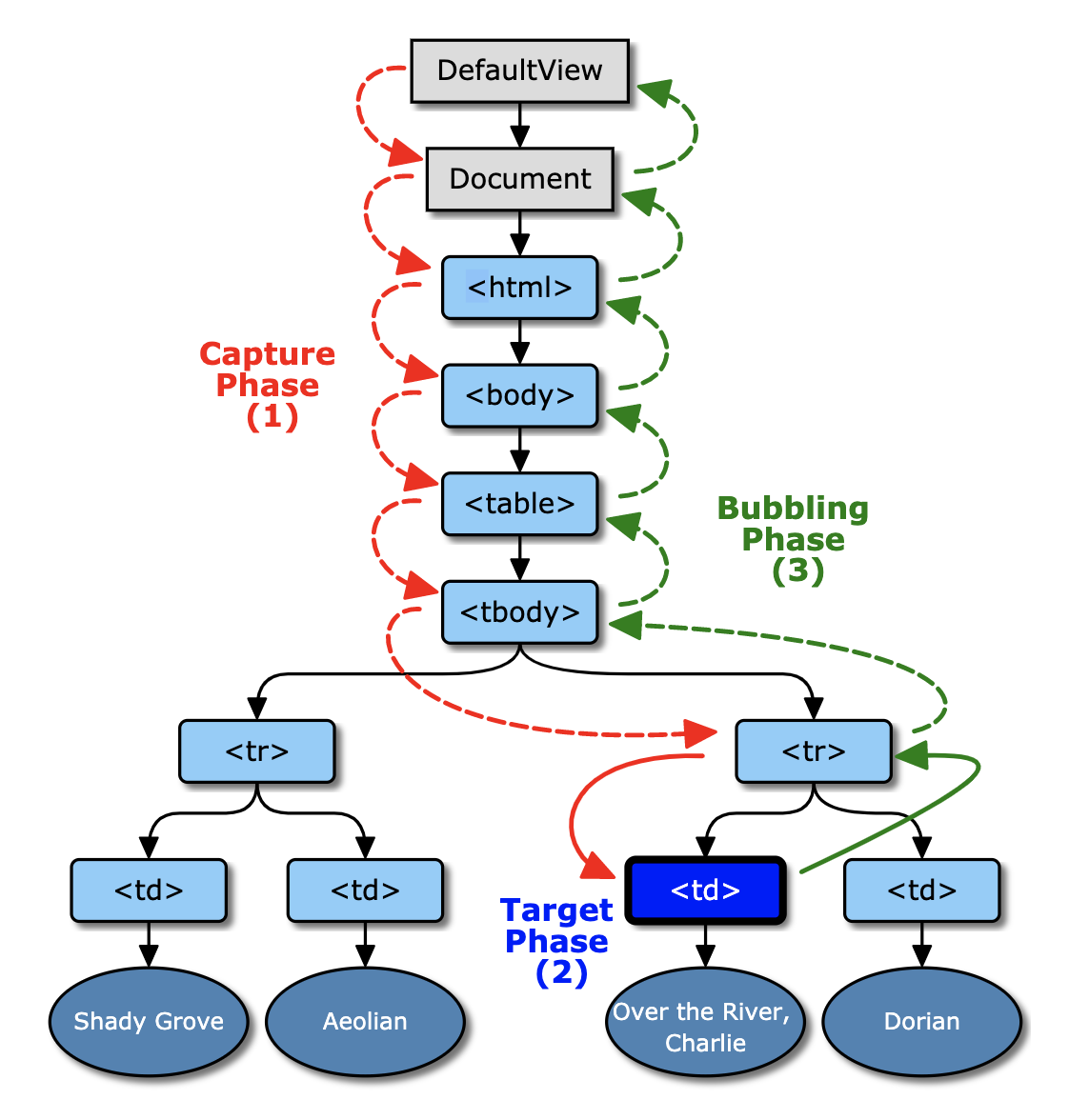
}

button.addEventListener('click', handleClick) // Створення обробника

button.removeEventListener('click', handleClick) // Видалення обробника

**!** Обробники подій можна видаляти тільки тоді, коли вони були додані за допомогою **addEventListener**

Події. **Етапи аналізу**



1. **Capturing Phase** (Спуск) або фаза захоплення

element.addEventListener('click', myFunction, **true**) // Захоплення

В цій фазі подія аналізується від найстаршого батьківського елемента і знижується вниз до елемента, на якому виникла подія. Це означає, що обробник подій, призначений для батьківського елемента, викликається перед обробником дочірнього елемента.

1. **Target Phase** (фаза цільового елемента)  
   Подія досягає цільового (спрацьовуючого) елемента, на якому виникла подія, і тут викликається обробник події для цього елемента.
2. **Bubbling Phase** (Підняття) фаза всплеску або здійснення (**Значення за замовчуванням**)

element.addEventListener('click', myFunction, **false**) // Всплеск або здійснення

Після обробки фази цільового елемента подія починає вспливати вгору від цільового елемента до найстаршого батьківського елемента. Обробники подій, призначені для батьківських елементів, викликаються відповідно до порядку вкладеності.

За замовчуванням більшість обробників подій додаються в фазі всплеску (bubbling phase), оскільки вона є більш звичайною. Аргумент useCapture можна встановити як true, якщо потрібно, щоб обробник подій виконувався в фазі захоплення (capturing phase).

Об'єкт **Event**

Об'єкт **Event** - використовується для представлення події, яка відбулася з елементом. Event містить різноманітну інформацію про подію, таку як тип події, елемент, на якому виникла подія, координати в момент події, тощо.

Об'єкт Event є базовим об'єктом для різних типів подій, таких як події клавіатури (keydown, keyup), миші (click, mousemove), форм (submit, change), та багатьох інших.

document.getElementById('myButton').addEventListener('click', function(event) {

console.log('Тип події:', **event.type**)

console.log('Елемент, на якому виникла подія:', **event**.**target**)

console.log('Координати події:', **event**.**clientX**, **event**.**clientY**)

})

Властивості об’єкта **Event**

Об'єкт **Event** містить багато властивостей, які надають інформацію про подію, яка виникла. Ось деякі з них:

1. **type** Рядок, який вказує тип події, наприклад, 'click', 'keydown', 'submit' і т.д.
2. **target** Елемент DOM, на якому виникла подія.
3. **currentTarget** Елемент DOM, на якому встановлений обробник подій (зазвичай це той самий елемент, що і target, але може відрізнятися в контексті використання фази захоплення).
4. **timeStamp** Таймстемп (час в мілісекундах) в момент виникнення події.
5. **eventPhase** Число, яке вказує на фазу обробки події. Може бути одним з наступних значень: Event.NONE, Event.CAPTURING\_PHASE, Event.AT\_TARGET, Event.BUBBLING\_PHASE.
6. **bubbles** Логічне значення, яке показує, чи підтримує ця подія механізм всплеску (bubbling).
7. **cancelable** Логічне значення, яке показує, чи може подія бути скасована за допомогою методу preventDefault().
8. **defaultPrevented** Логічне значення, яке показує, чи була викликана методом preventDefault() для цієї події.
9. **isTrusted** Логічне значення, яке вказує, чи є ця подія дійсно виниклою (true), чи штучно створеною (false).
10. **keyCode** або **key** Ці властивості містять інформацію про натиснуту клавішу при події клавіатури.
11. **layerX, layerY** Горизонтальна та вертикаль позиція курсора миші
12. **which** Номер натиснутої кнопки миші
13. **modifiers** Чи натиснута клавіша Модифікатор (Alt, Ctr, Shift). Бітова маска, одна з констант ALT\_MASK, CONTROL\_MASK, SHIFT\_MASK, and META\_MASK
14. **data** Масив з рядків, який містить url об’єктів, які були переглянуті та впущені у вікно
15. **pageX, pageY** Горизонтальна та вертикаль позиція курсора відносно веб сторінки
16. **screenX, screenY** Горизонтальна та вертикаль позиція курсора на екрані
17. **width, height** Ширина та висота вікна після зміни розмірів

Відміна подій **preventDefault()**

Процес, коли обробник події вирішує припинити (перервати) потік виконання алгоритму події.

Наприклад, ви можете відмінити подію натискання клавіші, щоб недопустити введення невалідних символів в поле вводу.

const myInput = document.getElementById('myInput')

const errorText = document.getElementById('errorText')

myInput.addEventListener('input', function(event) {

const inputValue = this.value

const invalidChars = /[!@#$%^&\*()\_+{}\[\]:;<>,.?~\\/]/

if (invalidChars.test(inputValue)) {

event.**preventDefault**() // Відмінюємо введення невалідних символів

errorText.textContent = 'Введення невалідних символів заборонено!'

} else {

errorText.textContent = ' '

}

})

Для відміни дії браузера на подію, також можна додати елементу атрибут onclick = ‘return false’

<a href = ‘Any.html’ onclick = ‘return false’> Link </a>

**Bubling** - Підйом подій

**Підйом подій (event bubbling)**   
Під час цієї фази браузер перебирає елементи, починаючи з цільового (найнижчого елемента), і піднімається вгору по DOM-дереву до кореня (найвищого елемента). Кожен елемент на цьому шляху може викликати свій власний обробник подій.

Це означає, що якщо ви призначили обробник подій для батьківського елемента, і ви клікнули на дочірній елемент, то спочатку виконається обробник для дочірнього елемента, а потім для батьківського, якщо такий існує.

Підйом подій є стандартною поведінкою браузера і використовується за замовчуванням. Це дозволяє створювати ієрархічну структуру обробки подій і логічно групувати їх. Наприклад, ви можете створити обробники подій для батьківських елементів, що впливають на всі дочірні елементи в межах цього батьківського елемента.

Делегування обробки події

**Делегування обробки подій** - це підхід в програмуванні, коли ви встановлюєте один обробник подій на **батьківський елемент** замість встановлення окремих обробників на кожен **дочірній елемент**. Цей підхід дозволяє зменшити кількість обробників подій, спростити код та ефективно взаємодіяти з багатьма елементами, особливо коли елементи додаватимуться або змінюватимуться динамічно.

<ul id="**myList**">

<li>Елемент 1</li>

<li>Елемент 2</li>

<li>Елемент 3</li>

</ul>

<script>

const **myList** = document.getElementById(**'myList'**)

**myList**.addEventListener('click', function(event) {

if (**event.target.tagName** === 'LI') {

console.log('Клікнуто на:', event.target.textContent);

}

})

</script>

**Event**  - Генерування подій

**Генерування подій** - це процес штучного спрацьовування подій на елементах DOM (Document Object Model) в веб-розробці. Такий механізм, дозволяє викликати обробку певної події на елементі без фактичного фізичного взаємодії користувача з цим елементом.

    <button id='myButton'>Go</button>

    <script>

      function f1() {

        alert('Вітаємо на сайті!')

      }

      myButton.onclick = f1 // При натисканні користувачем на button «Go» – виклич функцію f1

const **clickEvent** = **new** **Event**(**'click'**, { // **Генеруємо** **нову (Штучну) подію - клік**

bubbles: true, // дозволяє події вспливати вгору

cancelable: true, // можна скасувати подію

view: window, // вікно, з якого спрацьовує подія

})

// імітуємо настання події myButton.onclick

myButton.**dispatchEvent**(clickEvent) // надсилаємоо подію «**клік**» елементу myButton

  </script> // В результаті користувач відразу отримає - alert(' Вітаємо на сайті!')

**CustomEvent** - Генерування користувацьких подій

**CustomEvent** є класом у JavaScript, який дозволяє створювати та використовувати власні користувацькі події з визначеними даними. Цей клас використовується для створення подій, які не є стандартними DOM-подіями.   
Відмінності CustomEvent від звичайних подій (таких як click, mouseover тощо) полягають у тому, що можна передавати додаткові дані в об'єкті події, що робить їх більш гнучкими для специфічних потреб.

const customEvent = new CustomEvent('customEventName', { // Створюємо користувацьку подію "customEventName"

detail: { // Об`єкт, який містить додаткові дані

message: 'Hello, custom event!',

value: 42

},

bubbles: true, // чи має вспливати подія

cancelable: true // чи можна скасувати подію

})

const targetElement = document.getElementById('myElement') // Отримуємо елемент

targetElement.addEventListener('customEventName', function(event) { // Додаємо обробник події до елементу

console.log(event.detail.message) // "Hello, custom event!"

console.log(event.detail.value) // 42

})

targetElement.dispatchEvent(customEvent) // Запускаємо нашу користувацьку подію на обраному елементі

У цьому прикладі ми створили CustomEvent з ім'ям "customEventName" і передали об'єкт detail, який містить додаткові дані, які ми можемо отримати в обробнику події. Після цього ми додали обробник події до елементу і запустили нашу користувацьку подію на цьому елементі.

CustomEvent дозволяє створювати і використовувати події, які відповідають специфічним потребам додатку чи бібліотеки, і передавати додаткові дані в обробники цих подій.

**Успадкування** Прототипи

Успадкування в JavaScript відноситься до механізму, за допомогою якого один об'єкт може успадковувати властивості та методи іншого об'єкта. Це є однією з основних частин об'єктно-орієнтованого програмування, що дозволяє створювати ієрархії класів та розширювати функціональність існуючих об'єктів.

В JavaScript успадкування досягається за допомогою прототипів.

**Прототип** - це об'єкт, в якому зберігаються властивості та методи, які спільні для багатьох об'єктів. Кожен об'єкт має прототип, і він може успадковувати властивості та методи свого прототипу.

Особливості:

* Прототип використовується тільки як резервне сховище властивостей для ЧИТАННЯ (якщо у об’єктів немає власної властивості, то цю властивість шукаємо у об’єкті прототипі (об’єкті, адреса якого міститься у властивості \_\_proto\_\_) )
* Операції додавання нових властивостей і видалення властивостей виконуються виключно над самим об’єктом
* При зміні значення властивості (яку читали з прототипу) у об’єкті створюється власна копія цієї властивості (і надалі використовуємо власну копію)

**Створення об’єктів на основі прототипів Застарілі способи**

* З використанням властивості **\_\_proto\_\_**  **Капєц який старий спосіб**  
  За допомогою \_\_proto\_\_ виконується додавання прототипа до **існуючого** об’єкта.

      let obj = {       // Створюємо об’єкт

        prop\_1: 11,

        meth\_1: function () {

          document.write('meth\_1')

        }

      }

      let objParent = {       // Створюємо предка

        prop\_2: 22,

        meth\_2: function () {

          document.write('meth\_2')

        }

      }

obj.\_\_proto\_\_ = objParent // Додаємо предка у \_\_proto\_\_

* З використанням методу **Object.create()**

Object.create( прототип ) **с**творює **новий** об’єкт з вказаним прототипом.

let objParent = {       // Створюємо предка

prop\_2: 22,

        meth\_2: function () {

          document.write('meth\_2')

        }

    }

let obj = Object.create(objParent) // Створюємо об’єкт на основі прототипу

obj.prop\_1 = 7 // Додаємо створеному об’єкту властивість

    obj.meth\_1 = function () { // Додаємо створеному об’єкту метод

          document.write('meth\_1')

     }

Object.create( прототип, descriptions ) **с**творює **новий** об’єкт з вказаним прототипом, і додає нові властивості та методи вказані за допомогою дескриптора

let objParent = {       // Створюємо предка

prop\_2: 22,

  meth\_2: function () {

    document.write('meth\_2')

    }

}

let obj = Object.create(objParent, { // Створюємо об’єкт на основі прототипу

prop\_1: { // Додаємо створеному об’єкту властивість

value: 11,

enumerable: true

},

method\_1: { // Додаємо створеному об’єкту метод

value: function () {

          document.write('meth\_1')

     },

enumerable: true

}

})

**extends** Створення об’єкта на основі прототипу **Сучасний спосіб**

class ChildClass extends ParentClass {

// Конструктор та методи підкласу

}

* ChildClass - це підклас або клас-спадкоємець, який успадковує властивості та методи з ParentClass.
* ParentClass - це батьківський клас, властивості та методи якого успадковуються підкласом.

class Animal {

constructor(name) {

this.name = name

}

speak() {

console.log(`${this.name} makes a sound.`)

}

}

class Dog extends Animal {

constructor(name, breed) {

**super**(name) // Виклик конструктора батьківського класу

this.breed = breed

}

speak() {

**super**.speak() // Виклик методу батьківського класу

}

}

const myDog = new Dog("Buddy", "Golden Retriever")

console.log(myDog.name) // "Buddy"

console.log(myDog.breed) // "Golden Retriever"

myDog.speak() // "Buddy makes a sound."

Встановлення **нового прототипу**  **Оbject.setPrototypeOf**( obj, prototype )

Встановлення **нового прототипу** для вже **існуючого об'єкта**

Метод, який дозволяє динамічно додавати новий прототип для вже існуючого об'єкта. Проте, зауважте, що використання Object.setPrototypeOf() може призвести до погіршення продуктивності і може бути не найкращим рішенням в більш складних структурах даних.

const animal = {

type: "Unknown",

makeSound: function() {

console.log("Some sound")

}

}

const dog = {

type: "Dog",

bark: function() {

console.log("Woof woof!")

}

}

console.log(dog.type) // Виведе "Dog"

dog.bark() // Виведе "Woof woof!"

dog.makeSound() // Виникне помилка, бо об'єкт dog не успадковує makeSound

Object.setPrototypeOf(dog, animal) // Змінюємо прототип dog на об'єкт animal

dog.makeSound() // Тепер виведе "Some sound", бо dog успадковує makeSound від animal

Повернення прототипу об'єкта **Оbject.getPrototypeOf**(obj)

Метод, який повертає прототип об'єкта

const animal = {

type: "Unknown",

makeSound: function() {

console.log("Some sound")

}

}

const dog = Object.create(animal) // dog успадковує властивості та методи від animal

const dogPrototype = Object.getPrototypeOf(dog) // Отримання прототипу об'єкта dog

console.log(dogPrototype === animal) // Виведе true, бо прототип dog є animal

**prototypeObj.isPrototypeOf**(object)

Метод, який перевіряє, чи є об'єкт одним з прототипів іншого об'єкта

const animal = {

type: "Unknown",

makeSound: function() {

console.log("Some sound")

}

}

const dog = Object.create(animal) // dog успадковує властивості та методи від animal

console.log(animal.isPrototypeOf(dog)) // Виведе true, бо animal є прототипом для dog

console.log(dog.isPrototypeOf(animal)) // Виведе false, бо animal не успадковується від dog

**obj.hasOwnProperty**(propertyName)

Метод, який використовується для перевірки, чи має об'єкт певну **власну** властивість (property). **Власні властивості** - це ті, які прямо присутні в об'єкті, і вони не успадковуються з прототипу об'єкта.

let person = {

name: "John",

age: 30

}

console.log(person.hasOwnProperty("name")) // **true**, оскільки "name" - це власна властивість об'єкта person

console.log(person.hasOwnProperty("gender")) // **false**, оскільки "gender" – об'єкт person не має такої властивості

**Object.getOwnPropertyNames**(obj)

Метод, який використовується для отримання масиву **імен всіх власних** властивостей (включаючи незалежно від того, чи вони перераховуються, чи ні) об'єкта в форматі рядків. Цей метод не включає в себе імена властивостей, що успадковані від прототипу об'єкта.

let person = {

name: "John",

age: 30

}

let propertyNames = Object.getOwnPropertyNames(person)

console.log(propertyNames) // Виведе: ["name", "age"]

**Object.getOwnPropertyDescriptor**(obj, propertyName)

Метод, який використовується для отримання дескриптора властивості об'єкта. Дескриптор властивості містить інформацію про характеристики властивості, такі як **її значення**, можливість зчитування (**readable**), можливість запису (**writable**), перераховуваність (**enumerable**) та можливість налаштування (**configurable**).

let person = {

name: "John"

}

let descriptor = Object.getOwnPropertyDescriptor(person, "name");

console.log(descriptor) // {

value: "John",

writable: true,

enumerable: true,

configurable: true

}

Метод Object.getOwnPropertyDescriptor повертає об'єкт, який містить властивості, які відображають дескриптор властивості.

* value: Значення властивості.
* writable: Логічне значення, яке вказує, чи можна змінювати значення властивості.
* enumerable: Логічне значення, яке вказує, чи властивість буде перераховуваною в циклах for...in.
* configurable: Логічне значення, яке вказує, чи можна змінювати дескриптор властивості або видаляти властивість.

**Object.defineProperty**(obj, propertyName, descriptor)

Метод, який використовується **для додавання** нової властивості в об'єкті **або зміни характеристик** існуючої властивості. Цей метод надає контроль над властивостями об'єкта, включаючи можливість зчитування (**readable**), можливість запису (**writable**), перераховуваність (**enumerable**) та можливість налаштування (**configurable**).

obj - об'єкт, для якого ви хочете визначити властивість.

propertyName - рядок, що представляє ім'я властивості, яку ви хочете визначити або змінити.

descriptor - об'єкт, який містить опис властивості, включаючи характеристики, такі як **value**, **writable**, **enumerable** та **configurable**.

let person = {}

Object.defineProperty(person, "name", {

value: "John",

writable: true,

enumerable: true,

configurable: true

})

console.log(person.name) // "John"

* value: Значення властивості.
* writable: Логічне значення, яке вказує, чи можна змінювати значення властивості.
* enumerable: Логічне значення, яке вказує, чи властивість буде перераховуваною в циклах for...in.
* configurable: Логічне значення, яке вказує, чи можна змінювати дескриптор властивості або видаляти властивість.

Дескриптор властивості може містити методи **get** і **set**, які визначають поведінку для зчитування і запису властивості. Коли ви використовуєте методи **get** і **set** в дескрипторі властивості, ця властивість вважається обчислювальною (computed property).

let person = {

firstName: "John",

lastName: "Doe",

**get** fullName() {

return this.firstName + " " + this.lastName;

},

**set** fullName(value) {

let parts = value.split(" ")

this.firstName = parts[0]

this.lastName = parts[1]

}

}

console.log(person.fullName) // Зчитування обчислювальної властивості

person.fullName = "Alice Johnson" // Запис обчислювальної властивості

console.log(person.firstName) // "Alice"

console.log(person.lastName) // "Johnson"

**Object.defineProperties**(obj, descriptors)

Метод, який використовується для додавання або зміни **декількох** властивостей об'єкта одночасно. Він дозволяє вказати конфігурацію для кожної властивості за допомогою об'єкта дескрипторів.

let person = {};

Object.defineProperties(person, {

name: {

value: "John",

writable: true,

enumerable: true,

configurable: true

},

age: {

value: 30,

writable: true,

enumerable: true,

configurable: true

}

})

console.log(person.name) // "John"

console.log(person.age) // 30

**Object.preventExtensions**(obj)

Метод, який використовується **для заборони додавання нових** властивостей до об'єкта. Після використання цього методу неможливо буде додати нові властивості до об'єкта, але можна змінювати існуючі властивості та їх значення, та видаляти існуючі властивості.

**Object.isExtensible**(obj)

Метод, який використовується **для перевірки**, чи можна розширювати (додавати) нові властивості до об'єкта. Він повертає логічне значення **true**, якщо до об'єкта можна додавати нові властивості, і **false**, якщо додавання нових властивостей заборонено.

let person = {

name: "John",

age: 30

}

console.log(Object.isExtensible(person)) // **true**

Object.preventExtensions(person)

console.log(Object.isExtensible(person)) // **false**

person.name = "Alice" // Можна змінювати існуючі властивості

person.gender = "Female" // Не можна додати нову властивість

console.log(person) // { name: "Alice", age: 30 }

**Object.seal**(obj)

Метод, який використовується для запечатування об'єкта. Коли об'єкт запечатаний, ви **не можете** **додавати** нові властивості до нього, **видаляти** існуючі властивості або змінювати тип деяких властивостей з об'єкта.

Після використання Object.seal, всі властивості об'єкта залишаються доступними для зчитування і запису, але ви не можете додати нові властивості або змінювати або видаляти існуючі властивості. Однак **значення властивостей можна змінювати**.

**Object.isSealed**(obj)

Метод, який використовується **для перевірки**, чи є об'єкт запечатаним. Запечатаний об'єкт - це об'єкт, до якого можна звертатися для зчитування і запису існуючих властивостей, але не можна додавати нові властивості або видаляти існуючі.

Метод Object.isSealed повертає логічне значення **true**, якщо об'єкт запечатаний, і **false**, якщо об'єкт не запечатаний.

let person = {

name: "John",

age: 30

}

console.log(Object.isSealed(person)) // **false**

Object.seal(person)

console.log(Object.isSealed(person)) // **true**

person.age = 35 // Зміна значення властивості - **дозволено**

person.gender = "Male" // Додавання нової властивості - **не дозволено**

delete person.name // Видалення властивості - **не дозволено**

**Object.freeze**(obj)

Метод, який використовується для заморожування (фіксації) об'єкта. Заморожений об'єкт є статичним, і ви не можете змінювати його властивості або їх значення. Жодна властивість не може бути додана, видалена або змінена після застосування Object.freeze.

**Object.isFrozen**(obj)

Метод, який використовується для перевірки, чи є об'єкт замороженим. Метод Object.isFrozen повертає логічне значення **true**, якщо об'єкт заморожений, і **false**, якщо об'єкт не заморожений.

let person = {

name: "John",

age: 30

}

console.log(Object.isFrozen(person)) // **false**

Object.freeze(person)

console.log(Object.isFrozen(person)) // **true**

person.age = 35 // Зміна значення властивості - не дозволено

person.gender = "Male" // Додавання нової властивості - не дозволено

delete person.name // Видалення властивості - не дозволено

**try / catch** Обробка виключних ситуацій

У випадку, якщо у програмі виникає ситуація, коли подальше штатне виконання є неможливим (як правило – це критична помилка), то у програмі генерується виключна ситуація. Для обробки виключних ситуацій може бути використано блок **try/catch**

// -- Розробляємо кастомні помилки --

      class IsNotNumberError extends Error { // У випадку, якщо значення не є числом

        constructor() {

          super()

          this.message = 'Має бути числом'

          this.name = 'IsNotNumberError'

        }

      }

      class IsNegativeNumberError extends Error { // У випадку, якщо значення є від"ємним

        constructor() {

          super()

          this.message = "Не може бути від'ємним"

          this.name = 'IsNegativeNumberError'

        }

      }

      class IsTooHightNumberError extends Error { // У випадку, якщо число є дуже великим

        constructor(currentNumber) {

        super()

          this.currentNumber = currentNumber

          this.message = 'Не може бути більшим за 65'

          this.name = 'IsTooHightNumberError'

        }

      }

      function validate() {

        try {

          var exp = document.forms[0].elements[0].value

          if (isNaN(exp)) throw new IsNotNumberError()

          if (exp < 0) throw new IsNegativeNumberError()

          if (exp > 65) throw new IsTooHightNumberError(exp)

        } catch (err) {

          if (err instanceof IsNotNumberError) {

            console.log(err.message) // Має бути числом

            document.forms[0].elements[0].value = 0

          } else if (err instanceof IsNegativeNumberError) {

            console.log(err.message) // Не може бути від'ємним

            document.forms[0].elements[0].value = 0

          } else if (err instanceof IsTooHightNumberError) {

            console.log(err.message) // Не може бути більшим за 65

            document.forms[0].elements[0].value = 65

          } else console.log(err.message)

          return false

      }

        return true

      }

Об'єкт **Date**

Об'єкт Data використовується для роботи з датами та часом. Він дозволяє створювати, змінювати і відображати дати та час у програмному коді. Date є вбудованим об'єктом у JavaScript і надає багато методів для роботи з датами.

Створення об'єкта **Date**

Створення об'єкта **Date**, без аргументів (у цьому випадку він буде ініціалізований поточною датою та часом)

const currentDate = new Date() // Поточна дата та час

Створення об'єкта **Date**, з одним параметром - кількість мілісекунд, що пройшли з початку 1970 року.

Ціле число, вказане в якості мілісекунд, називається міткою часу (**timestamp**).

const date = new Date(0) // 01.01.1970 UTC+0

const Jan02\_1970 = new Date(24 \* 3600 \* 1000) // 02.01.1970 UTC+0  
const Dec31\_1969 = new Date(-72 \* 3600 \* 1000) // 29.12.1969 UTC+0

Створення об'єкта **Date,** з єдиним аргументом (**рядок**)

const date = new Date("2017-01-26") // 01.26.2017 UTC+0

Створення об'єкта **Date,** з декількома параметрами (від 2х)

const customDate = new Date(2023, 8, 2, 12, 30, 0) // 2 вересня 2023, 12:30:00

new Date(**year, month, date, hours, minutes, seconds, ms**)

* **year -** має містити 4 цифри. Для сумісності також допускаються 2 цифри, які вважаються 19xx, напр. 98 тут те саме, що й 1998, але настійно рекомендується завжди використовувати 4 цифри.
* **month**  - рахунок місяців починається з 0 (січня), до 11 (грудня).
* **date**  - день місяця, якщо він відсутній, то береться 1.
* Якщо **hours**/**minutes**/**seconds**/**ms** відсутні, вони вважаються рівними 0.

Отримання компонентів **Date**

Повернення різних компонентів дати і часу, таких як рік, місяць, день, година, хвилина, секунда та інші, використовуючи наступні методи

const date = new Date() // Поточна дата та час – Sat Sep 02 2023 15:39:38 GMT+0300

const year = date.getFullYear() // **2023** Отримуємо рік поточної дати

const year = date.get**UTC**FullYear() // **2023** Отримуємо рік поточної дати за Грінвічем

const month = date.getMonth() // **8** (вересень) **!** Отримуємо місяць поточної дати (від 0 до 11)

const month = date.get**UTC**Month() // **8** Отримуємо місяць поточної дати за Грінвічем

const date = date.getDate() // **2** Отримуємо день місяця поточної дати

const date = date.get**UTC**Date() // **2**  Отримуємо день місяця поточної дати за Грінвічем

const hour = date.getHours() // **15** Отримуємо години поточного часу

const hour = date.get**UTC**Hours() // **12** Отримуємо години поточного часу за Грінвічем

const minute = date.getMinutes() // **39** Отримуємо хвилини поточного часу

const minute = date.get**UTC**Minutes() // **39** Отримуємо хвилини поточного часу за Грінвічем

const second = date.getSeconds() // **38** Отримуємо секунди поточного часу

const second = date.get**UTC**Seconds() // **38** Отримуємо секунди поточного часу за Грінвічем

const second = date.getMilliseconds() // **710** Отримуємо мілісекунди поточного часу

const second = date.get**UTC**Milliseconds() // **710** Отримуємо мілісекунди поточного часу за Грінвічем

const day = date.getDay() // **6** Отримуємо день тижня (від 0 до 6)

const day = date.get**UTC**Day() // **6** Отримуємо день тижня (від 0 до 6) за Грінвічем

const day = date.getTime() // **1693649236710** Отримуємо **timestamp**

const day = date.getTimezoneOffset() // **-180** Отримуємо зміщення часової зони відносто UTC у **хвилинах**

Встановлення значень **Date**

Зміна дати і часу, за допомогою наступних методів

const date = new Date() // Поточна дата та час – Sat Sep 02 2023 13:33:43 GMT+0300

date.setFullYear(2024) // Змінено день тижня та рік – **Mon** Sep 02 **2024** 13:33:43 GMT+0300

date.setMonth(5) // Змінено день тижня та місяць – **Sun** **Jun** 02 2024 13:33:43 GMT+0300

date.setDate(15) // Змінено день тижня та місяця – **Sat** Jun **15** 2024 13:33:43 GMT+0300

date.setHours(14) // Змінено години - Sat Jun 15 2024 **14**:33:43 GMT+0300

date.setMinutes(45) // Змінено хвилини - Sat Jun 15 2024 14:**45**:43 GMT+0300

date.setSeconds(30) // Змінено секунди - Sat Jun 15 2024 14:45:**30** GMT+0300

date.setTime(1253649236710) // Змінено дату - **Tue Sep 22 2009 22:53:56** GMT+0300

date.setDate(268) // Змінено день тижня, місяця, місяць, рік – **Wed May 26 2010 22:53:56** GMT+0300

Перетворення **Date** у рядок

Об'єкт Date має різні методи для перетворення дати та часу в різні формати рядка або отримання рядкового представлення дати.

|  |  |
| --- | --- |
| const date = new Date() | // Поточна дата та час – Sat Sep 02 2023 13:56:30 GMT+0300 |
| date.toString() | // Повертає рядкове представлення дати та часу |
| date.toISOString() | // Повертає рядкове представлення дати та часу у форматі ISO 8601, який є міжнародним стандартом для представлення дати та часу. |
| date.toGMTString() | // Повертає рядкове представлення дати та часу у форматі GMT (Greenwich Mean Time) |
| date.toUTCString() | // Подібний до toGMTString() і повертає рядкове представлення дати та часу у форматі UTC (Coordinated Universal Time). |
| date.toJSON() | // Повертає дату та час у форматі JSON. Цей формат відповідає формату ISO 8601. |
| date.toDateString() | // Повертає рядкове представлення лише дати, без часу. |
| date.toTimeString() | // Повертає рядкове представлення лише часу, без дати. |
| date.toLocaleString() | // Повертає рядкове представлення дати та часу з урахуванням локалізації. Він використовує налаштування локалізації користувача, такі як мова та формат дати та часу. |
| date.toLocaleDateString() | // Повертає рядкове представлення лише дати, з урахуванням локалізації. |
| date.toLocaleTimeString() | // Повертає рядкове представлення лише часу, з урахуванням локалізації. |

Рядки String

**Рядки** – Незмінюваний тип даних !

**Рядки** - це один з фундаментальних типів даних в мові програмування JavaScript, і вони використовуються для зберігання і обробки текстової інформації в програмах.

Створення рядків може бути виконане кількома способами:

1. Створення з використанням лапок (кодові лапки).  
   Для оголошення рядків використовуються одинарні лапки (') або подвійні лапки (").

let myString = "Це рядок в JavaScript" // typeof(myString) поверне ‘string’

1. Створення з використанням конструктора **String**  
   Цей спосіб створення рядка використовує конструктор **String**, який створює об'єкт рядка. Важливо відзначити, що більшість рядкових операцій в JavaScript вимагають примітивних рядків, тому зазвичай використовуються прості лапки.

let myString = new String("Це ще один рядок в JavaScript.") // **typeof(myString) поверне ‘object’**

Ескейп послідовності

* Новий рядок тексту, при виконанні коду (**\n**)  
  Використовується для створення нового рядка або переходу до наступного рядка в тексті

let multiLineString = "Це перший рядок.\n Це другий рядок." //

* Новий рядок коду, при написанні коду (**\**)  
  Використовується для перенесення рядка коду

let multiLineString = "Це перший рядок.\

Це другий рядок."

* Табуляція (**\t**)  
  Використовується для вставки символу табуляції в рядок. Це може бути корисно для форматування тексту або таблиць

let indentedText = "Текст з табуляцією:\n\tТекст зі зміщенням на табуляцію."

* Екранування символів – якщо потрібно використовувати метасимволи як літерали, їх можна екранувати за допомогою **\**, наприклад, **\**\, **\**.

document.write(‘c: **\**\doks**\**\files’)

Використовуються для вставки одинарних або подвійних лапок всередину рядка

let singleQuoteString = 'Це рядок з одинарною лапкою: **\**'одинарна лапка**\**'.'

let doubleQuoteString = "Це рядок з подвійною лапкою: **\**"подвійна лапка**\**"."

* Unicode Escape (**\u**XXXX)  
  Використовується для вставки символу Unicode в рядок. XXXX - це чотири шістнадцяткові цифри, які представляють код символу Unicode

let unicodeString = "Це символ Unicode для літери 'A': \u0041"

Метод **repeat()**

Створення рядка, у вигляді багаторазової копії створеного фрагмента

let hello = ‘hello\_’

console.log(hello.repeat(3)) // hello\_hello\_hello

Методи **padStart()**  **padEnd()** -Заповнення рядків

Методи padStart() і padEnd() є частиною специфікації ECMAScript 2017 (ES8) і призначені для додавання символів до початку або в кінець **рядка (! Стрінги)** відповідно так, щоб досягти певної заданої довжини.

* padStart(maxLength, fillString)

Цей метод додає символ заповнювач (вказані як fillString) до початку **рядка** до того моменту, поки рядок не досягне або перевищить задану довжину (maxLength). Якщо вихідний рядок вже має або перевищує задану довжину, метод не вносить змін і повертає вихідний рядок.

let myNumber = 123

let myStringNumber = myNumber.toString()

let paddedNumber = myStringNumber.padStart(5, "0") // Результат: "**00**123"

* padEnd(maxLength, fillString)

Цей метод додає символи (вказані як fillString) до кінця рядка до того моменту, поки рядок не досягне або перевищить задану довжину (maxLength). Якщо вихідний рядок вже має або перевищує задану довжину, метод не вносить змін і повертає вихідний рядок.

let text = **"**123**"**

let paddedText = text.padEnd(5, "0") // Результат: "123**00**"

**Задача**. Дано масив імен. Вирівняти довжину усіх імен, за довжиною найбільшого імені

      let arr = ['John', 'Bob', 'Ali', 'Sandra']

      let maxLength = Math.max(...arr.map((name) => name.length))

      let res = arr.map((name) => name.padStart(maxLength, '-'))

      document.write(res) // --John, ---Bob, ---Ali, Sandra

Використання шаблону рядка у функції

Дозволяє передати у функцію-шаблон шаблонний рядок ,при цьому в цю функцію передаються усі динамічно обчислювані фрагменти як окремі параметри. Також, при вставці значень у шаблон, можливо виконувати над ними деякі операції.

**Задача**. Дано ім’я користувача, стать і вік. З використанням шаблонів вивести на екран рядок за шаблоном

      `Привіт, {ім'я} тобі {к-сть років} років`

function greeting(part,name, age) { // Функція-шаблон

return `part[0] ${name} part[1]${age} part[2]`

}

var userName = "Іван" // Динамічні дані

var userAge = 30

var message = greeting**’**Привіт, ${userName}! тобі ${userAge} років.**’** // Виведе "Привіт, Іван! Тобі 30 років."

**Задача**. Дано ім’я користувача, стать і вік. З використанням шаблонів вивести на екран рядок за шаблоном

      `{пан/пані} **\_** {ім’я} **:** {пенсіонер/не пенсіонер}`

    function getMessage(fixedParts, gender, name, age) { // Функція-шаблон

        const genderLabel = gender === 'm' ? 'Пан' : 'Пані'

        const pensionerTitle =

          (gender === 'm' && age >= 65) || (gender === 'w' && age >= 60)

            ? 'пенсіонер' : 'не пенсіорен'

        return `${genderLabel} ${fixedParts[0]} ${name.toUpperCase()} ${fixedParts[1]} ${pensionerTitle}`

      }

      let userName = 'Ivan' // Динамічні дані

      let gender = 'm'

      let age = 31

      let message = getMessage**’**${gender} **\_** ${userName}  **:** ${age}**’** // Виведе "Пан**\_**IVAN **:** не пенсіонер"

Звертання до символів рядка

* Звертання за індексом

let str = "Hello"

let character = str[0] // Отримати перший символ ("H")

* charAt()

let str = "Hello"

let character = str.charAt(0) // Отримати перший символ ("H")

* Звертання до коду символа

let str = "Hello"

let code = str.charCodeAt(0) // Отримати код першого символа (72)

let character = String.fromCharCode(code) // Перетворити код на символ ("H")

**Задача**. Користувач вводить логін. Перевірити правильність логіна('winner') (регістр не враховується)

      const correctLogin = 'winner'

      const login = prompt('Login')

      // if (login.toLowerCase() === correctLogin.toLowerCase()) alert('Ok')

      // else alert('Error')

      if (login.toUpperCase() === correctLogin.toUpperCase()) alert('Ok')

      else alert('Error')

**Задача**. Шифр Цезаря.

      function getCesarCode(letter, shift) {

        const letterCode = letter.charCodeAt(0)

        const minLetterCode = 'A'.charCodeAt(0) // код літери 'A' - 65

        const maxLetterCode = 'Z'.charCodeAt(0) // код літери 'Z' - 91

        const newCode =

          ((letterCode - minLetterCode + shift) % (maxLetterCode - minLetterCode + 1)) + minLetterCode

        const newCharacter = String.fromCharCode(newCode)

        return newCharacter

      }

      let message = prompt('Message')

      let newMessage = [].map

        .call(message, (letter) => getCesarCode(letter, 2))

        .join('')

      document.write(newMessage)

Методи об'єднання рядків

* Використання оператора **+**

let str1 = "Hello"

let str2 = "World"

let result = str1 + " " + str2 // Результат: "Hello World"

* Використання методу **concat()**

let str1 = "Hello"

let str2 = "World"

let result = str1.concat(" ", str2) // Результат: "Hello World"

Методи вибору підрядка **substring() substr() slice()**

* Використання **substring()**

Метод substring(start, end) витягує підрядок від індексу start (включно) до індексу end (не включаючи). Якщо не вказано end, витягує до кінця рядка.

let str = "Hello, World!"

let substring = str.substring(0, 5) // Результат: "Hello"

Якщо використати метод substring() без вказівки end, він автоматично витягне підрядок до кінця рядка.

let str = "Hello, World!"

let substring = str.substring(7) // Результат: "World!"

* Використання **substr()**

Метод substr(start, length) витягує підрядок, починаючи з індексу start та маючи довжину length.

var str = "Hello, World!";

var substring = str.substr(0, 5) // Результат: "Hello"

* Використання **slice()**

Метод slice(start, end) діє схоже на substring(). Він витягує підрядок від індексу start (включно) до індексу end (не включаючи). Якщо не вказано end, витягує до кінця рядка.

let str = "Hello, World!"

let substring = str.slice(0, 5) // Результат: "Hello"

Методи substring() і slice() в JavaScript використовуються для отримання підрядка з рядка, але вони мають кілька важливих відмінностей:

* Параметри:

substring(start, end): Приймає два параметри, start і end. Він повертає підрядок, що починається з індексу start і закінчується перед індексом end. Якщо start більше, ніж end, то substring() автоматично обмінює їх значеннями перед виділенням підрядка.

slice(start, end): Також приймає два параметри, start і end. Він повертає підрядок, який починається з індексу start і закінчується перед індексом end. Якщо start більше, ніж end, то slice() повертає пустий рядок.

* Обробка від'ємних значень:

substring() визначає від'ємні значення start і end як 0. Тобто, якщо ви передаєте від'ємні значення для start або end, substring() розглядає їх як 0.

slice() дозволяє використовувати від'ємні значення для start або end, що вказує на відлік від кінця рядка. Наприклад, slice(-3) видасть останні 3 символи рядка.

* Поведінка без аргументів:

substring() і slice() повертають копію всього рядка, якщо ви не передаєте аргументів. Тобто, substring() і slice() без аргументів повернуть копію вихідного рядка.

* Поведінка з одним аргументом:

Якщо передати лише один аргумент в substring() або slice(), вони поведуться схоже: substring(start) і slice(start) вирізають підрядок, що починається з індексу start до кінця рядка.

Отже, хоча substring() і slice() дозволяють отримувати підрядки з рядків, вони мають різні особливості щодо обробки аргументів і від'ємних значень. Вибір між ними залежить від конкретних потреб вашого завдання.

Методи пошуку елементів у рядку

* рядок**.indexOf(**підрядок,startIndex**)**

Метод indexOf() знаходить перший індекс входження підрядка substring у рядок. Якщо підрядок не знайдено, повертається -1. Ви також можете вказати початковий індекс пошуку через startIndex.

let str = "Hello, World!"

let index = str.indexOf("World") // Результат: 7

* рядок**.lastIndexOf(**підрядок,startIndex**)**

Метод lastIndexOf() знаходить останній індекс входження підрядка у рядок, починаючи з кінця рядка. Якщо підрядок не знайдено, повертається -1. Ви також можете вказати початковий індекс пошуку через startIndex.

let str = "Hello, World!";

let lastIndex = str.lastIndexOf("o") // Результат: 8

* рядок**.includes(**підрядок**)**

Метод includes() перевіряє, чи містить рядок підрядок і повертає true або false.

var str = "Hello, World!"

var isPresent = str.includes("World") // Результат: true

* рядок**.startsWith(**підрядок**)** ірядок**.endsWith(**підрядок**)**

Метод startsWith() перевіряє, чи починається рядок з підрядка.

Метод endsWith() перевіряє, чи закінчується рядок підрядком. Вони повертають true або false.

var str = "Hello, World!"

var startsWithHello = str.startsWith("Hello") // Результат: true

var endsWithWorld = str.endsWith("World!") // Результат: true

**Задача**. Дано адресу сайту. Перевірити, чи є він з шифруванням (починаєтсья з «https»)

let url = '\_\_https://test.test'

if (url.trim().toLowerCase().startsWith('https')) alert('ok')

else alert('Danger')

**Задача**. Дано адресу сайту. Перевірити, чи закінчується шлях з «edu».

let url = 'https://test.test.edu'

f (url.endsWith('edu')) alert('ok')

else alert('Danger')

Порівняння рядків

Порівняння рядків у JavaScript відбувається на основі лексикографічного порівняння, тобто порівнюються символи в рядках зліва направо. Результат порівняння рядків визначається у вигляді булевого значення (true або false).

* == (рівність) та != (нерівність)

Ці оператори порівнюють рядки на рівність без врахування типів даних.

"abc" == "abc" // true

"abc" != "def" // true

5 == "5" // true

* === (строго рівність) та !== (строго нерівність)

Ці оператори порівнюють рядки за значенням і типом даних. Вони повертають true, якщо рядки ідентичні за значенням і типом, і false в іншому випадку.

"5" === 5 // false, оскільки рядок і число мають різні типи.

* > (більше), < (менше), >= (більше або рівне), <= (менше або рівне)

Ці оператори порівнюють рядки за лексикографічним порядком (алфавітним)

"apple" > "banana" // false, оскільки лексикографічно "apple" менше "banana".

* localeCompare()

Цей метод порівнює два рядки і повертає числове значення, яке вказує на їхнє відносне положення в алфавітному порядку. Він є більш потужним для міжнародних порівнянь, оскільки враховує локалізацію.

* String.prototype.localeCompare():

Цей метод дозволяє порівнювати рядки з врахуванням регіональних налаштувань (локалізації). Він приймає інший рядок для порівняння та опції, які дозволяють налаштувати порівняння відповідно до потреб конкретної локалізації.

Методи зміни фрагментів у рядку **replace() replaceAll()**

Заміна підрядків в рядку іншими підрядками.

* **replace()** приймає два аргументи: підрядок (або регулярний вираз) для пошуку і підрядок, який буде вставлений вмісці знайденого підрядку.  
  Виконує перший знайдений збіг і повертає новий рядок із заміненим підрядком.  
  Заміна відбувається тільки один раз.

var str = "Це гарний день, гарний день!"

var newStr = str.replace("гарний", "прекрасний") // Виведе "Це прекрасний день, гарний день!"

* **replaceAll()** приймає два аргументи: підрядок (або регулярний вираз) для пошуку і підрядок, який буде вставлений вмісці всіх знайдених підрядків.

Виконує заміну всіх збігів і повертає новий рядок із всіма заміненими підрядками.  
Заміна відбувається для всіх входжень, а не тільки для першого.

var str = "Це гарний день, гарний день!";

var newStr = str.replaceAll("гарний", "прекрасний") // Виведе "Це прекрасний день, прекрасний день!"

Видалення пробілів **trim()**

Методи trim(), trimStart(), trimLeft(), trimEnd(), і trimRight() в JavaScript використовуються для видалення пробілів і інших пробільних символів з початку та (або) кінця рядка.

let str = " Привіт, як справи? "

* **trim()**

Видаляє пробіли та інші пробільні символи з обох кінців рядка і повертає новий рядок.  
Цей метод є самим загальним і часто використовується для очищення введення користувача або даних з зовнішніх джерел.

let trimmed = str.trim() // Виведе "Привіт, як справи?"

* **trimStart()** або **trimLeft()**

Видаляє пробіли та інші пробільні символи тільки з початку рядка і повертає новий рядок.  
Цей метод корисний, коли вам потрібно видалити пробіли з початку рядка, але залишити їх на кінці.

let leftTrimmed = str.trimStart() // Виведе "Привіт, як справи? "

* **trimEnd()** або **trimRight()**

Видаляє пробіли та інші пробільні символи тільки з кінця рядка і повертає новий рядок.  
Цей метод корисний, коли вам потрібно видалити пробіли з кінця рядка, але залишити їх на початку.

let rightTrimmed = str.trimEnd() // Виведе " Привіт, як справи?"

Об'єднання та розділення рядків **split() join()**

Метод split() дозволяє розділити рядок на масив рядків, а join() дозволяє об'єднати масив рядків у один рядок.

* **split()**

Цей метод використовується для розділення рядка на підрядки за заданим роздільником і створення масиву із отриманими підрядками.

string.split(separator, limit)

separator: Це рядок або регулярний вираз, за яким буде проведено розділення. Рядок буде розділяти за кожним входженням цього роздільника.

limit (необов'язковий): Це число, яке обмежує кількість елементів у вихідному масиві.

var str = "яблуко,банан,киві"

var fruits = str.split("**,**") // Виведе масив ["яблуко", "банан", "киві"]

* **join()**

Цей метод використовується для об'єднання всіх елементів масиву в один рядок, розділеній заданим роздільником.

array.join(separator)

separator: Це рядок, який буде використовуватися як роздільник між елементами масиву при об'єднанні.

var fruits = ["яблуко", "банан", "киві"]

var str = fruits.join(", ") // Виведе рядок "яблуко, банан, киві"

**Регулярні вирази**

Регулярні вирази (англ. regular expressions або regex) - це потужний інструмент для обробки тексту, пошуку та витягування інформації з рядків на основі шаблонів. Вони використовуються в багатьох програмах та мовах програмування для виконання різних завдань обробки тексту, таких як пошук, заміна, валідація, аналіз та інші. Задання регулярного виразу може відбуватися двома способами:

* Задання регулярного виразу за допомогою **літералу**

let regexLiteral = /pattern/modifiers

Шаблон (pattern) – це регулярний вираз. Цей спосіб зручний і зазвичай використовується, коли регулярний вираз є сталим і не змінюється під час виконання програми.

Модифікатори (Modifiers) - це опціональна частина регулярного літералу, яка дозволяє налаштувати поведінку регулярного виразу. Модифікатори пишуться безпосередньо після закриваючого слешу ("/")

* **і** – ігнорування регістру.
* **g** – знаходження усіх фрагментів, які відповідають шаблону (пошук не зупиниться після знаходження першого фрагменту)
* **m** – пошук у багаторядковому тексті
* Створення **об'єкта регулярного виразу**

Дозволяє динамічно будувати шаблони на основі змінних. Цей підхід використовується, коли ви хочете конструювати регулярні вирази на льоту, наприклад, на основі даних, які отримані від користувача або з бази данних, тощо.

let pattern = "dynamicPattern"

let regexObject = new RegExp(pattern)

dynamicPattern - це динамічно створений рядок, який може містити регулярний шаблон. Ви можете пізніше змінювати pattern і перестворювати об'єкт regexObject, щоб відповідати новим шаблонам.

**Методи** регулярних виразів

Методи регулярних виразів - це функції (методи), які надаються бібліотеками для роботи з регулярними виразами і дозволяють виконувати різні операції з текстом на основі регулярних виразів.

* Метод **test**   
  Перевіряє, чи відповідає заданий регулярний вираз тексту. Він повертає **true**, якщо текст відповідає регулярному виразу, і **false** - в іншому випадку. Цей метод найчастіше використовується для перевірки наявності регулярного виразу в тексті без витягування конкретного текстового фрагмента.

let text = "Hello FROM Me"

let result = /from/**i**.test(text) // Перевіряє, чи є "from" у тексті - поверне **true**

* Метод **exec**   
  Шукає перший фрагмент, який відповідає заданому регулярному виразу у тексті і повертає інформацію про це співпадіння у вигляді об'єкта. Якщо співпадіння не знайдено, метод повертає null.

let text = "Hello FROM Me"

let result = /from/**i**.exec(text) // Перевіряє, чи є "from" у тексті

// *Console****[****'FROM', index: 6, input: 'Hello FROM Me', groups: undefined****]***

**0**: "FROM" // Дав фрагмент, який співпав з регулярним виразом

**groups**: undefined

**index**: 6 // Індекс, з якого починається знайдений фрагмент

**input**: "Hello FROM Me" // Рядочок, в якому він шукав

**length**: 1 // Скільки таких фрагментів знайдено

[[Prototype]]: Array(0)

**Методи** рядків

Методи рядків у регулярних виразах використовуються для взаємодії з рядками і виконання операцій, пов'язаних з регулярними виразами.

let str = 'fdd 23 ras 33 f56 33 fdsss3 111'

let useTest = /[0-9]/.test(str) // true

let useExec = /33/.exec(str)

// *Console*

*['33', index: 11, input: 'fdd 23 ras 33 f56 33 fdsss3 111', groups: undefined]*

**0**: "33"

**groups**: undefined

**index**: 11

**input**: "fdd 23 ras 33 f56 33 fdsss3 111"

**length**: 1

[[Prototype]]: Array(0)

* Метод **search**   
  Використовується **для пошуку першого входження** заданого регулярного виразу у тексті і повертає позицію (індекс) початку співпадіння або -1, якщо співпадіння не знайдено.
* Метод **match**   
  Використовується **для витягування всіх співпадінь** заданого регулярного виразу з тексту. Він повертає масив, в якому кожен елемент є текстовим фрагментом, що відповідає регулярному виразу, або null, якщо співпадінь не знайдено. Якщо регулярний вираз має групи, цей метод також поверне значення груп

let useSearch = **str**.search(/[0-9]/) // Поверне 4 – індекс початку співпадіння

let useMatchWith-g = **str**.match(/[0-9]/**g**) // ['2', '3', '3', '3', '5', '6', '3', '3', '3', '1', '1', '1']

let useMatch = **str**.match(/[0-9]/) // ['2', index: 4, input: 'fdd 23 ras 33 f56 33 fdsss3 111', groups: undefined]

* Метод **replace**   
  Дозволяє замінювати текст, що відповідає регулярному виразу, іншим текстом або результатом функції. Ви можете використовувати цей метод для заміни всіх співпадінь або першого співпадіння в рядку.

const text = "The price is $10.99 and $20.00."

const regex = /\$\d+\.\d{2}/g

const replacedText = text.replace(regex, "X.XX")

console.log(replacedText) // Виводить "The price is X.XX and X.XX."

**Правила** опису Регулярних виразів (шаблонів)

* **Літерали**  
  Включають звичайні символи, такі як літери та цифри, які відповідають собі самим. Наприклад, регулярний вираз /apple/ відповідає тексту, в якому є слово "apple".
* **Метасимволи**  
  Це спеціальні символи, які мають особливе значення в регулярних виразах.

**.** (крапка) **–** довільний символ за вийнятком кінця рядка або нового рядка

text = 'caat, cirt, axcaet'

text.match(/c**..**t/g) //  ['caat', 'cirt', 'caet']

**\d –** довільна цифра (від 0 до 9)

**\D –** не цифра, теж саме що [^0-9]

**\w –** буква (латинського алфавіту), цифра, символ нижнього підкреслювання

**\W –** не буква, не цифра, не символ нижнього підкреслювання

**\b** - слово починається або закінчується вказаним символом або групою

**\B** - слово містить в собі, або закінчується вказаним символом або групою.

**\s –** пробільний символ

**\S**  – не пробільний символ

**\n –** символ нового рядка

**\t –** символ табуляції

**|** (вертикальна риска) - використовується для вказівки альтернативи між можливими співпадіннями.

**() –** використовується для створення груп символів.

* **Символьні класи**   
  Дозволяють вказувати множину символів, які можуть відповідати.

[abdtf] - будь-яка мала літера з вказаних (перерахованих)

[a-z] - будь-яка мала літері від "a" до "z".

[0-9] - будь-яка цифра від 0 до 9.

[A-Za-z] - будь-яка літера (велика або мала).

[^0-9] - будь-який символ, крім вказаних цифр.

* **Квантифікатори**   
  Вказують кількість разів, які символ або група можуть повторюватися.

{n} - рівно n разів.

{n,} - принаймні n разів або більше.

{n, m} - від n до m разів (включно).

**\*** **–** 0 або більше входжень (є або не має) попереднього символу або групи.

let str = ‘Hellooo Word! Hello W3Schools!’

let result = str.match(/lo\*/g) // l,looo,l,l,lo,l

**+**  **–** 1 або більше входжень попереднього символу або групи.

let str = ‘Hellooo Word! Hello W3Schools!’

let result = str.match(/o+/g) // ooo,o,o,oo

**? –** 0 або 1 входжень попереднього символу або групи.

* **Позиція фрагмента у пошуку**   
  Вказують на певну позицію у тексті.

^ - вказує на фрагмент, який знаходиться на початку рядка.

$ - вказує на фрагмент, який знаходиться в кінці рядка.

**Робота з групами** у регулярних виразах

Робота з групами у регулярних виразах дозволяє виділити та зберегти підстроки, які відповідають певним шаблонам у тексті. Групи створюються, при використанні круглих дужок () у регулярному виразі для об'єднання частин шаблону у логічні блоки.

* Створення групи і отримання підстроки з групи

const text = "Name: John, Age: 30"

const regex = /Name: (\w+), Age: (\d+)/

const match = regex.exec(text)

const name = match[1] // Отримуємо "John" з першої групи

const age = match[2] // Отримуємо "30" з другої групи

* Зміна тексту з використанням груп

const date = "2023-09-13"

const regex2 = /(\d{4})-(\d{2})-(\d{2})/

const formattedDate = date.replace(regex2, "$2/$3/$1") // Замінюємо на "09/13/2023"

* Звернення до групи в середині регулярного виразу

let str = “He said: \”She’s the one!\”.”

let reg = /([‘ “])(.\*?)**\1**/g // Поверне ”She’s the one!”

* Іменовані групи в регулярних виразах  
  Це спосіб надати ім'я групам у регулярних виразах, що робить їх більш читабельними та дозволяє легше отримувати доступ до результатів по їхнім іменам. Щоб створити іменовану групу, ви використовуєте синтаксис (?<ім'я>вираз)

let html = ‘<input type="text" name="username" value="john\_doe">

<input type="email" name="email" value="john@example.com">’

// Створюємо регулярний вираз для вилучення атрибутів input

let regex = /<input type="(?<inputType>[^"]+)" name="(?<inputName>[^"]+)" value="(?<inputValue>[^"]+)">/g

let match

while ((match = regex.exec(html))) {

let inputType = match.groups.inputType

let inputName = match.groups.inputName

let inputValue = match.groups.inputValue

console.log(`Type: ${inputType}, Name: ${inputName}, Value: ${inputValue}`);

}

// Поверне Type: text, Name: username, Value: john\_doe

Type: email, Name: email, Value: [john@example.com](mailto:john@example.com)

* Незапам'ятовуючі групи  
  Незапам'ятовуючі групи, створюються за допомогою **?:** після відкриваючої дужки **(** у регулярному виразі.

const text = "apple, banana, cherry"

const regex = /(**?:**\w+, )+\w+/

const match = text.match(regex) // Виводить ["apple, banana, cherry"]

У цьому прикладі (?:\w+, ) - це незапам'ятовуюча група. Вона відповідає послідовності слова, коми та пробілу, але не запам'ятовує цю послідовність як окреме співпадіння. Результатом є масив із одним співпадінням, яке містить весь рядок "apple, banana, cherry", а не окремі слова.

* (**?<=** фрагмент)   
  Ця група дозволяє знайти текст, який знаходиться після певного фрагмента, але без включення цього фрагмента у співпадіння.

const text = "Hello, world!"

const regex = /(?<=Hello, )\w+/

const match = text.match(regex) // Виводить "world"

У цьому прикладі (Hello, ) є фрагментом, і ми шукаємо слово, яке йде після цього фрагмента.

* (**?<!** фрагмент)   
  Ця конструкція дозволяє знайти текст, який не має певного фрагмента перед ним.

const text = "apple, banana, cherry";

const regex = /\w+(?!, )/;

const match = text.match(regex) // Виводить "apple"

У цьому прикладі (, ) є фрагментом, і ми шукаємо слова, які не мають цього фрагмента перед собою, використовуючи негативну зворотню відповідність.

* (**?=** фрагмент)   
  Ця конструкція дозволяє знайти текст, який містить певний фрагмент перед собою.

const text = "123abc";

const regex = /(?=\d+)\w+/;

const match = text.match(regex) // Виводить "abc"

У цьому прикладі (?=\d+) є фрагментом, і ми шукаємо слово, яке має цей фрагмент перед собою, використовуючи позитивну впередню відповідність.

* (**?!** фрагмент)   
  Ця конструкція дозволяє знайти текст, який не містить певного фрагмента перед собою.

const text = "123abc"

const regex = /(?=\d+)\w+/

const match = text.match(regex) // Виводить "abc"

У цьому прикладі (?!\d+) є фрагментом, і ми шукаємо слово, яке не містить цього фрагмента перед собою, використовуючи негативну впередню відповідність.

Жадібний (greedy) та лінивий (lazy) пошук

Це два режими квантифікаторів у регулярних виразах, які визначають, як вони спробують відповідати тексту

* **Жадібний пошук**  
  Якщо ви використовуєте жадібний пошук, то регулярний вираз /".+"/ спробує знайти найбільший можливий фрагмент, який заключений у подвійні лапки.

const text = 'a "witch" and her "broom" is one'

const regex = /".+"/

const match = text.match(regex) // **“**"witch" and her "broom"**”**

* **Лінивий пошук**  
  Зараз давайте спробуємо лінивий пошук, використовуючи регулярний вираз /".\*?"/. Лінивий пошук спробує знайти найменший можливий фрагмент, який заключений у подвійні лапки

const text = 'a "witch" and her "broom" is one'

const regex = /".\*?"/

const match = text.match(regex) // **"**witch**"**, **"**broom**"**