## Основи синтаксису

Скрипт - це послідовність команд (або файл-сценарій), які автоматизовують виконання деяких задач.

***Підключення (додавання) скриптів***

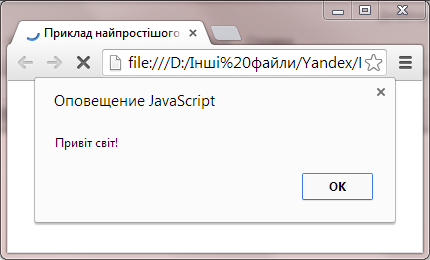
Скрипт можна додати в будь-яке місце HTML документу, за допомогою елемента script. Стандарти HTML4 вимагають обов'язкового вказування атрибуту type= "text/javascript" (у HTML5 таких вимог вже немає). Є кілька способів для додавання скипту на веб сторінку.

***1) безпосереднє включення в текст сторінки.***

Для цього оператори записують безпосередньо між тегами

|  |  |
| --- | --- |
| ***Загальна форма*** | ***Приклад*** |
| <scrip type="text/javascript">скрипт</script> | <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8">  <title>Приклад найпростішого скрипта</title>  **<script >**  **alert("Привіт світ!")**  **</script>**  </head>  <body>  Скрипт виконано.  </body>  </html> |

Для його запуску необхідно створити html сторінку, що містить наступний текст і відкрити за допомогою браузера.



***2) підключення окремих файлів з скриптами (мають розлишення \*.js)***

Для цього необхідно створити текстовий файл, що містить скрипт і зберегти його з розширенням js. А на сторінку він підключається за допомогою елемента script з атрибутом src.

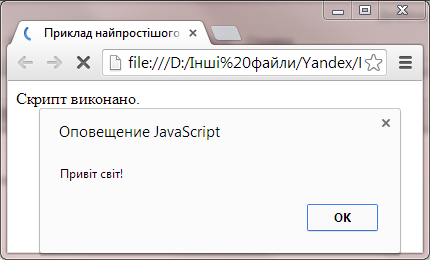
|  |  |
| --- | --- |
| Загальна форма | <script type="text/javascript" src="шлях/ім'я.js"> </script> |
| Приклад | ----------- Файл Hello.js---------  alert("Привіт світ!");  ----------- Сторінка HTML ---------  <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8">  <title>Приклад найпростішого скрипта</title>  **<script src="Hello world.js"> </script>**  </head>  <body>  Скрипт виконано.  </body>  </html> |

<body>

Скрипт виконано.

<script type="text/javascript" src="…../Hello world.js"></script>

</body>



Якщо документ містить кілька скриптів, то вони виконуються послідовно, в порядку розміщення на сторінці.

Оператори розділяються за допомогою символу ";". Перехід на новий рядок не розділяє оператори, але при виконанні скрипта інтерпретатор намагається виправити помилки і може сам додати розділення, але це не завжди приводить до правильного результату.

***3)порядок виконання скриптів***

Загалом, виконання скрипта здійснюється одразу після завантаження файлу, що його містить. При цьому поки скрипт не буде виконано завантаження сторінки буде призупинено. У зв’язку з цим часто скрипти розміщують в кінці документу. Але іноді такий підхід є неприйнятним, оскільки інформація, яка знаходиться на сторінці може залежати від результатів виконання скрипта. Тому з метою деблокування процесу завантаження сторінки можна використовувати при підключенні зовнішніх скриптів атрибуту async

<script **async** src="шлях до скрипта">

Таким чином дається команда на завантаження скрипта (він одразу буде виконаний після завнтаження) і продовжується обробка веб-сторінки.

Але якщо у програмі підлючається декілька файлів скриптів, то виконання кожного наступного здійснюється після закінчення виконання попереднього. Тому, наприклад, при підлюченні двох сриптів у порядку

<script src="1.js" ></script>

<script src="2.js" ></script>

Виконаним буде спочатку скрипт 1 і лише після закінчення його буде завантажуватися скрипт 2. Якщо використати атрибут async,

<script src="1.js" **async**></script>

<script src="2.js" **async**></script>

то виконаним буде той скрипт, який завантажиться раніше.

Іноді потрібно дотриматися порядку виконання скриптів і при цьому не блокувати процес обробки сторінки браузером. В цьому випадку необхідно використати атрибут defer.

<script src="1.js" defer></script>

<script src="2.js" defer></script>

В цьому блокування сторінки не буде і першим обов’язково буде виконано перший скрипт.

Слід також зазначити, що скрипти, які підключені з використанням атрибуту defer виконуються після того, як весь HTML документ буде опрацьовано браузером.

Приклад.

<script src="defer2.js" defer></script>

<script src="async1.js" async></script>

Скрипт async1.js буде буде виконано як тільки він буде завантажений. Скрипт defer2.js буде виконано після опрацювання документа браузеном.

**Типи даних**

У JS підтримуються :

* прості типи даних (відповідають скалярним значенням: числам, логічним значенням та ін.);
* складні типи даних посилання (об’єкти, функції).

|  |  |
| --- | --- |
| number | як цілі так і дробові, мають тип Number |
| string | текстові дані (як окремі символи, так і рядки) |
| boolean | логічні значення начення ( true (істина) і false (неправда)) |
| undefined | спеціальне значення, яке використовується як значення змінної, якій не присвоєно значення |
| object | тип даних посилання, використовується при описі об’єктів  (null - спеціальне значення, яке використовується для позначення «порожньої адреси») |

Для визначення типу даних може бути використана функцію **typeof**, яка повертає назву типу даних (рядок тексту)

typeof undefined // "undefined"

typeof 0 // "number"

typeof true // "boolean"

typeof "foo" // "string"

typeof function myFunc(){}   //"function"

typeof {} // "object"

typeof null // "object"

typeof  {name:'John', age:34} // "object"

typeof  [1,2,3,4]             // "object"

Інші підходи до визначення типів даних будуть розглянуті пізніше.

**Коментарі**

*Коментар –* невиконувана частина тексту програми, що ігнорується компілятором і служить для вставки деяких поміток у програмі тільки для програміста. Коментарі бувають однорядковими та багаторядковими. *Однорядковий коментар* починається з символів «//» і закінчується у кінці рядка. Тобто всі символи до кінця рядка вважаються коментарем.

Приклад.

// Це однорядковий коментар \_

*Багаторядковий коментар* починається з символів «**/\***» і закінчується символами «**\*/**».

Приrлад.

**/\*** Це

багаторядковий

коментар **\*/**

*Літерали*

*Літерал* – це явно вказане значення декого типу. Розрізняють такі типи літералів:

|  |  |  |
| --- | --- | --- |
| *тип літерала* | *опис* | *приклади* |
| масив |  | *var a=[]*  *var b=[23,12,89]* |
| цілого типу | у десятковій системі числення – звичне нам ціле число | 125, -89, 108; |
| дійсного типу | у форматі з фіксованою крапкою – у записі числа є крапка, що розділяє цілу і дробову частити | 25.69, 2.0, 145.058 |
| у форматі з плаваючою крапкою – у записі використано символ «е» або «Е», що розділяє мантису від порядку | 3.5е5, 3е12, 678е2 |
| символьний | заданий у явному вигляді рядок у одинарних або подвійних лапках | var a=”Hello”  var b = “Peace”  var c=”” |
| прості ескейп-послідовності – службові символи починаються з символу «\» | \n -перехід на новий рядок,  \t -горизонтальна табуляція,  \’ - апостороф, і т. д. |
|  | ` … текст … ${ значення } … текст ` | // Інтерполяція рядків  var name = "Бобби", time ="сегодня";  `Привет ${name}, как ты ${time}?`  “Привет” +name+”, как ты”+time |
| логічний | може приймати два значення | true,  false |
| літерал об’єкта |  | var a={}  var b={name=”Ivan”} |
| порожньої адреси (нульовий літерал) | використовується у випадку, коли покажчик не містить жодної адреси | null |

| **Специальные символы в JavaScript** | |
| --- | --- |
| **Символ** | **Значение** |
| \b | Возврат (Backspace) |
| \f | Перевод или прогон страницы (Form feed) |
| \n | Перевод строки (New line) |
| \r | Возврат каретки (Carriage return) |
| \t | Табуляция (Tab) |
| \v | Вертикальная табуляция (Vertical tab) |
| \' | Апостроф или одинарная кавычка |
| \" | Двойная кавычка |
| \\ | Обратная косая черта (Backslash) |
| \*XXX* | Символ в кодировке Latin-1, представленный тремя восьмеричными числами *XXX* от 0 до 377. Например, \251 (символ ©). |
| \x*XX* | Символ в кодировке Latin-1, представленный двумя шестнадцатеричными числами *XX* от 00 до FF. Например, \xA9 (символ ©). |
| \u*XXXX* | Символ в Unicode, представленный четырьмя шестнадцатеричными числами *XXXX*. Например, \u00A9 (символ ©). |
| \u*{XXXXX}* | Символ в UTF-32BE. Например, \u{2F804} обозначает то же, что обычная запись \uD87E\uDC04. |

## Опис змінних

Змінна – іменована область пам’яті, призначена для збереження певних даних, які можуть змінюватися. Для використання змінних їх слід описати (оголосити).

Опис може бути здійснено декількома способами:

* як глобальна змінна (як властивість window);
* за допомогою var;
* за допомогою let.

***Опис змінних, як властивостей об’єкта window***

У попередніх версіях JS дозволялось створювати змінні лише задаючи їх ім’я і значення.

|  |  |
| --- | --- |
| Загальна форма | Приклад |
| змінна = значення; | a = 90; // рівносильно window.a=90 |

Це по суті призводило до створення відповідної властивості об’єкта **window** (глобальної змінної) і могло призвести до конфлікту імен у різних скриптах. Тому на даний момент такий підхід не використовуються, а при використанні сучасних стандартів (використанні режиму "use strict") -- призводить до помилки.

"use strict";

a = 90; // error: a is not defined

*Опис змінних за допомогою var*

|  |  |
| --- | --- |
| Загальна форма | Приклад |
| var змінна; | var a;  var b,c; |

При оголошенні також можна задавати початкове значення змінної.

|  |  |
| --- | --- |
| Загальна форма | Приклад |
| var змінна = початкове значення; | var a=7;  var b= “Hello”;  a=”Hi” |

Ім'я кожної змінної повинно бути унікальним правильним ідентифікатором. Правильний ідентифікатор (назва змінної) на мові JS повинен містити тільки букви латинського алфавіту, цифри та символи "$" і "\_". При цьому ідентифікатор не може починатись з цифри і співпадати із зарезервованими словами.

Правильні ідентифікатори: MyID, identifier, x, $, abc123, my\_id, \_true.

Неправильні:

my id //містить пробіл

Ok! // містить недозволений знак оклику !

12th //починається з цифри

100 //починається з цифри

for //зарезервоване слово.

Стосовно змінних є декілька важливих зауважень:

* мова JS є регістро-чутливою (розрізняє ВЕЛИКІ та малі букви). Тому myid та MyID - це різні ідентифікатори;
* змінні оголошуються глобально ( вони доступні в будь-якому скрипті, що є на сторінці);
* змінні в JS не є типізованими ( не можна строго вказати тип змінної, вона може змінювати свій тип, залежно від значення).

***Приклад.***

var x; // x – змінна (значення undefined);

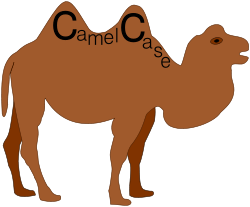
var y = 2;, у - змінна числового типу, рівна 2

x = y / 8; // х - числового типу, рівна 0,25

x = "String"; // змінює свій тип на рядковий.

***Вибір імені змінної***

**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 з

алежності від того, велика чи маленька перша літера:

* **UpperCamelCase** (*PascalCase*) (перша літера велика);
* **lowerCamelCase** (перша літера маленька).

Приклади імен: BackColor, backColor, CamelCase, itemCount, totalSum.

***Локальні змінні***

Іноді необхідно описати змінну, яка буде видима (доступна) тільки в межах блоку (в межах циклу, умовного оператора чи блоку, описаного з використанням фігурних дужок)

{

… блок …

}

Для цього необхідно описати ці змінні з використанням let

|  |  |
| --- | --- |
| Загальна форма | Приклад |
| {  let змінна;  } | {  let a;  a=23; 🡨 **Доступ можливий**  let b,c;  ...  a=a+5; 🡨 **Доступ можливий**  }  a=9; 🡨 **ПОМИЛКА! Змінна недоступна** |

## Опис констант

Константи описуються за допомогою оператора const.

|  |  |
| --- | --- |
| Загальна форма | Приклад |
| const КОНСТАНТА = значення; | const RATE=15; |

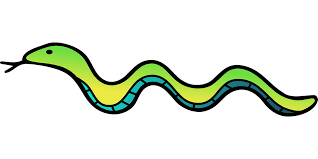
Константи, на відміну від змінних, не можуть змінювати свого значення. Традиційно імена констант записують великими літерами.

***Приклад.*** const PI = 3.14, ZERO = 0, NAME = "Name";

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

Приклад: **SCREAMING\_SNAKE\_CASE, MAX\_LENGTH, LOWER\_BOUND**

**const MAX\_LENGTH=100;**



## Введення та виведення даних

***Виведення даних***

Виведення даних може бути здійснено багатьма способами:

* виведення з використанням діалогового вікна **window.alert();**
* виведення в консоль браузера **console.log()**;
* виведення на сторінку HTML з використанням **document.write()**;
* виведення в HTML елемент, з використанням властивостей **innerHTML, innerText;**
* за допомогою форм.

Розглянемо деякі з них.

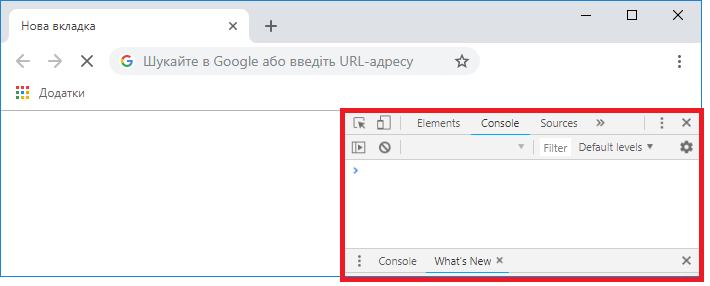
***Виведення даних за допомогою alert***

За допомогою виклику alert (або ж window.alert) відображаємо діалогове вікно, у якому і відображається вказана інформація.

|  |  |  |
| --- | --- | --- |
| Загальна форма | Приклад | На екрані |
| alert (текст повідомлення); | alert (“Hello”); |  |

|  |  |
| --- | --- |
| Загальна форма | alert (“ім’я змінної =”+змінна); |
| Приклад (вивести суму) | var a=9;  var b=7;  var sum=a+b;  **alert(“Sum=”+sum);** |
| На екрані |  |

***Виведення даних за допомогою console.log***



|  |  |
| --- | --- |
| Загальна форма | ***console.log*** (“текстове повідомлення”); |
| Приклад (вивести суму) | var a = 9;  var b = 7;  var sum = a + b;  **console.log("Sum ="+sum);** |
| На екрані |  |

***Виведення даних за допомогою document.write***

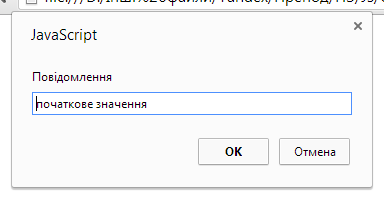
|  |  |
| --- | --- |
| Загальна форма | ***document.write*** (“текстове повідомлення”); |
| Приклад (вивести суму) | **document.write**("Приклад використання <br>");  var a = 9;  var b = 7;  var sum = a + b;  **document.write("Sum ="+sum);** |
| На екрані |  |

**Введення значення змінних**

Для введення даних будемо користуватись функцією prompt;

prompt (повідомлення, початкове значення);

В результаті виводиться модальне вікно, яке містить інформаційне повідомлення, поле для вводу з початковим значенням та кнопки Ok та Відміна. Зазвичай як другий параметр використовується порожній рядок. Якщо його не вказати то в деяких браузерах значення буде undefined.



Функція повертає значення, що було введено користувачем у відповідне поле. Значення повертається як текстова величина. Якщо необхідно ввести числове значення то необхідно скористатись функціями переведення parseInt, parseFloat чи оператором збереження знаку + . Детальніше вони будуть розглянуті в розділі про числовий тип. В разі натискання кнопки відміни функція повертає null.

*Введення цілого числа*

|  |  |
| --- | --- |
| *Загальна форма* | *Приклад* |
| var змінна =parseInt(prompt(“Змінна=”, “0”)) | var a= parseInt(prompt(“a=”, “0”)) |

*Введення дійсного числа*

|  |  |
| --- | --- |
| *Загальна форма* | *Приклад* |
| var змінна =parseFloat(prompt(“Змінна=”, “0”)) | var a= parseFloat(prompt(“a=”, “0”)) |

## Типи даних

### Числа

Всі числа в JavaScript, як цілі так і дробові, мають тип Number. Для запису числа використовуються арабські цифри. Для відділення дробової частини застосовується крапка.

В JavaScript можна записувати числа не тільки в десятковій, але і в шістнадцятковій (запис починається з 0x), а також вісімковій (починається з 0) системах числення: 0xFF це 255 в шістнадцятковій системі, а 010 - 8 в вісімковій системі. Також доступний запис «з плаваючою крапкою», який виглядає як мантиса e степінь. Наприклад: 1e3 - це 1\*103, тобто 1000, 3e-5 =3\*10-5=0.00003.

В пам’яті числа зберігаються в форматі IEEE-754, також відомому як «double precision». В ньому на число виділяється рівно 8 байт (64 біти). Деякі дробові числа в десятковому форматі мають скінченний запис, а в двійковій системі числення це нескінченна дріб. Так нескінченної дробом є 0.110=0,(00011)2. Двійкове значення нескінченних дробів зберігається лише до певного знака, тому виникає неточність. Переконатись в цьому можна на прикладі .

alert(0.1 + 0.2); // 0.30000000000000004, а не 0.3 як очікувалось

Втрата точності відбувається і з дуже великими числами:

alert(9999999999999999999999); //1e+22

Кожне число має метод toFixed([кількість знаків]), який округлює число до заданої точності і повертає результат у вигляді рядка. Кількість цифр після десяткового знак повинна знаходитися в діапазоні від 0 до 20 включно. Якщо не вказано, то вважається рівним 0. Число округлюється при необхідності, і дробова частина доповнюється нулями до потрібної довжини. Якщо число більше 1e +21, то метод повертає рядкове представлення в експоненціальному записі.

var n = 12.345;

alert (n.toFixed (1)); / / "12.3"

Викликати його можна не тільки у змінних числового типу, а й у числових літералів.

alert ((12.345).toFixed (1)); / / "12.3"

**Арифметичні вирази**

Арифметичний вираз – це вираз, в результаті обчислення якого одержуємо число. Над числами можна виконувати звичайні арифметичні та деякі спеціальні операції.

**Бінарні операції** (необхідно мати два операнди)

|  |  |
| --- | --- |
| x+y | Сума |
| x-y | Різниця |
| x\*y | Добуток |
| x/y | Частка |
| x%y | Остача від ділення |

**Унарні оператори (**необхідним є один операнд**)**

|  |  |
| --- | --- |
| -x | Зміна знаку на протилежний |
| +x | Збереження знаку. |
| x++,++x | Інкримент: збільшення на 1 |
| x--,--xx | Декримент: зменшення на 1 |

parseInt ('12px ') // 12

parseFloat ('12 .3.4') // 12.3

parseInt ('a123') // NaN

parseInt('0хFF') // 255

В JavaScript нема вбудованих математичних констант та функцій, але вони реалізовані як властивості та методи простору імен Math.

Основні методи:

|  |  |
| --- | --- |
| Math.floor (x) | Округлення вниз |
| Math.ceil (x) | Округлення вгору |
| Math.round (x) | Округлення до найближчого цілого |
| Math.acos (x) | Арккосинус x (в радіанах) |
| Math.asin (x) | Арксинус x (в радіанах) |
| Math.atan (x) | Арктангенс x (в радіанах) |
| Math.atan2 (y, x) | Арктангенс кута до точки (y, x). Виклик еквівалентний Math.atan (y / x). |
| Math.sin (x) | Синус x (в радіанах) |
| Math.cos (x) | Косинус x (в радіанах) |
| Math.tan (x) | Тангенс x (в радіанах) |
| Math.sqrt (x) | Квадратний корінь з x. |
| Math.log (x) | Натуральний логарифм x. |
| Math.pow (x, у) | Піднесення до степеня, повертає xу, наприклад Math.pow (2,3) = 8. |
| Math.abs (x) | Абсолютне значення числа |
| Math.exp (x) | Експонента ex. |
| Math.max (a, b, c ...) | Найбільший зі списку аргументів |
| Math.min (a, b, c ...) | Найменший зі списку аргументів |
| Math.random (x) | Псевдо-випадкове число в інтервалі [0,1) - тобто між 0 (включно) та 1 (не включаючи). Генератор випадкових чисел ініціалізуется поточним часом.   let min=10, max=78   let a =min + Math.floor(Math.random()\*(max-min+1)) |
| Math.PI | Число π |
| Math.Е | Константа Ейлера, е |

*Приклад.*  Скласти скрипт для генерування випадкового числа з проміжку [a,b];

var a=+prompt('a=',''), b=+prompt('b=','');

var rnd=a+Math.floor((b-a+1)\*Math.random());

alert(rnd);

### Логічний тип

Логічний тип даних Boolean має всього два значення:

* true («істина», «вірно» або ж «так»);
* false («неправда», «невірно» або ж «ні»).

Отримується в результаті обчислення логічних виразів (виконання операторів порівняння і ін.) та в умовах операторів if , while.

## Перетворення типів

Узагальнимо інформацію про систему перетворень типів у JavaScript.

***Перетворення у рядок (String)***

Рядкове перетворення відбувається, коли вимагається подання чого-небудь у вигляді рядка. Наприклад, його генерує функція alert. На практиці для явного перетворення часто застосовується оператор "+", у якого один з аргументів рядок. У цьому випадку він приводить до рядка і інший аргумент. Можна також здійснити перетворення явним викликом String(val).

***Перетворення у число (Number)***

Чисельне перетворення відбувається в математичних функціях і виразах, а також при нестрогому порівнянні даних різних типів. Для перетворення до числа в явному вигляді можна викликати Number(val) , або, покласти перед виразом оператор "+".

|  |  |
| --- | --- |
| **Значення** | **Перетвориться в ...** |
| undefined | NaN |
| null | 0 |
| true | 1 |
| false | 0 |
| Рядок | Пробіли по краях обрізаються. Далі один з випадків:  1) якщо залишається порожній рядок, то 0;  2) з непорожього рядка зчитується число;  3) якщо у рядку не число, то результат NaN . |

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

Перетворення до логічного типу (до true/false) відбувається у логічному контексті (там де небхідно перевірити виконання умови), такому як if(obj) , while(obj) і при застосуванні логічних операторів.

|  |  |
| --- | --- |
| **Значення** | **Перетвориться в ...** |
| undefined , null | false |
| Числа | Всі true , крім 0 , NaN - false . |
| Рядки | Всі true , крім порожнього рядка "" - false |
| Об'єкти | Завжди true |

***Логічні вирази***

*Логічний вираз*  -- це вираз в результати одержуємо одне із логічних значень (true/false).

Логічні вирази можуть містити:

* логічні константи (true/false);
* змінні логічного типу;
* оператори порівняння;

У JavScript використовуються такі оператори порівняння

|  |  |
| --- | --- |
| Більше | a > b |
| Менше | a < b . |
| Більше або дорівнює | a >= b |
| Менше або дорівнює | a <= b |
| Рівне  (a=b це присвоювання!) | a == b |
| Не рівно, ≠ | != |
| Тотожно рівно (співпадає тип і значення) | a === b |
| Не тотожно рівно (не співпадає тип, або не співпадають значення) | A !== b |

**При порівнянні за допомогою знаку “==” значення перетворюються до числа (до типу Number)**, крім випадку коли обидва значення є рядками. Порівняння рядків здійснюється у лексикографічному порядку (аналогічно, як ми упорядковуємо слова у алфавітному порядку).

Звичайна рівність (записана за допомогою знаку «==» ) не може відрізнити 0 від false. Оскільки і 0, і false при перетворенні у число дають 0 (а саме це відбувається при порівнянні за знаком «==»). Для перевірки рівності без перетворення типів використовуються оператори тотожної рівності === і !== . Вони порівнюють без приведення типів. Якщо тип різний, то такі значення завжди нерівні.

Для запису більш складних логічних виразів (умов) можна використати логічні операції:

&& -- логічне «і»;

|| -- логічне «або»;

! – логічне заперечення («не»).

Вони називаються «логічними», але в JavaScript можуть застосовуватися до значень будь-якого типу і повертають також значення будь-якого типу.

Розглянемо таблицю значень з результатами застосування даних операторів

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x | y | x&&y  (логічне «і») | x || y  (логічне «або») | !x  (заперечення) |
| false | false | false | false | true |
| false | true | false | true | true |
| true | false | false | true | false |
| true | true | true | true | false |

**КОРОТКИЙ ЦИКЛ ОБЧИСЛЕНЬ**

Логічне АБО в класичному програмуванні працює таким чином: «якщо хоча б один з аргументів true, то повертає true, інакше - false».

JavaScript обчислює кілька АБО зліва направо. При цьому, щоб економити ресурси, використовується так званий «короткий цикл обчислення».

Припустимо, обчислюються кілька АБО поспіль: a | | b | | c | | .... Якщо перший аргумент - true, то результат свідомо буде true (хоча б одне із значень - true), і інші значення ігноруються.

Це особливо помітно, коли вираз, передане в якості другого аргументу, має сторонній ефект - наприклад, присвоює змінну.

При запуску прикладу нижче привласнення x не відбудеться:

var x;

true || (x = 1); / / просто обчислимо АБО, без if

alert (x); / / undefined, x не присвоєно

А в прикладі нижче перший аргумент - false, так що АБО спробує обчислити другий, запустивши тим самим присвоювання:

var x;

false || (x = 1);

alert (x); / / 1

Оператор АБО обчислює рівно стільки значень, стільки необхідно - до першого true. Він повертає те значення, на якому зупинилися обчислення.

У класичному програмуванні && повертає true, якщо обидва аргументи істинні, а інакше – false.

До І застосуємо той же принцип «короткого циклу обчислень», але трохи по-іншому, ніж до АБО.  
Якщо лівий аргумент - false, оператор І повертає його і закінчує обчислення. Інакше - обчислює і повертає правий аргумент.

Оператор НЕ спочатку приводить аргумент до логічного типу true / false. Потім повертає протилежне значення.

*Використання короткого циклу обчислень для аналізу значення величини і задання значення за замовчуванням*

var b = parseInt(prompt("b=",""));

var c = b || 17; 🡨 Якщо значення змінної b некоректне, то значення змінної с=17

document.write("c="+c);

## Умовний оператор

Іноді, залежно від певних умов, потрібно виконати різні дії. Для цього використовується умовний оператор. Він може бути в повній

|  |  |
| --- | --- |
| Загальна форма | Приклад |
| **if** (умова)  оператор, якщо умова виконується  **else**  оператор, якщо умова не виконується | if(a>b)  max=a;  else  max=b; |

або скороченій формі.

|  |  |
| --- | --- |
| Загальна форма | Приклад |
| **if** (умова)  оператори, якщо умова виконується | max=a;  if(b>max)  max=b; |

Якщо при виконанні чи невиконанні деяких умов треба виконати більше ніж один оператор, то потрібно використовувати складений оператор (блок), який записується за допомогою фігурних дужок

{

оператор1;

оператор2;

. . . . .

}

|  |  |
| --- | --- |
| Загальна форма | Приклад |
| if (умова)  {  оператори, якщо умова виконується  }  else  {  оператори, якщо умова не виконується  } | if(a>b)  {  max=a;  min=b;  }  else  {  max=b;  min=a;  } |

або скороченій формі.

|  |  |
| --- | --- |
| Загальна форма | Приклад |
| if (умова)  {  оператори, якщо умова виконується  } | if(x!=0)  {  z=1/x;  alert(z);  } |

***Тернарний оператор***

Іноді буває ситуація, коли значення виразу повинно дорівнювати одному із двох значень у залежності від виконання чи невиконання деякої умови. Наприклад, потрібно залежно від умови присвоїти змінній певне значення. В цьому випадку можна використати тернарний оператор.

|  |  |
| --- | --- |
| Загальна форма | Змінна = умова ? значення1 (умова викон.) : значення2(умова не викон.);  ----- аналог з умовним оператором ------  if ( умова )  змінна = значення1;  else  змінна = значення2; |
| Приклад | **max= a>b ? a : b;**  ------ аналог з умовним оператором ------  if(a>b)  max=a;  else  max=b; |

*Приклад.*  Скласти скрипт для знаходження більшого з двох чисел.

var a=parseFloat(prompt('Введіть перше число',''));

var b=+prompt('Введіть друге число','');

var max=(a>b)?a:b;

alert('Більше число рівне '+max);

**ОПЕРАТОР ВИБОРУ SWITCH**

Конструкція switch замінює собою відразу кілька if. Це оператор викорситовується тоді, коли у залежності від значення деякої величини (кількість можливих значень є невеликою) потрібно виконати ті чи інші оператори.

Це більш наочний спосіб порівняти вираз (в сенсі оператора «===») відразу з декількома варіантами.

**switch**(селектор) {

**case** значення 1 : оператори   // if (селектор === значення1)

**break;**

**case** значення 2 : оператори // if (селектор === значення2)

**break;**

. . . . . . . . . . . . .

**case** значення n : оператори // if (селектор === значенняn)

**break;**

**default**: //інакше (всі інші значення)

оператори

}

|  |  |
| --- | --- |
| **Загальна форма** | **Приклад.** Вводиться оцінка – цифра, вивести оцінку прописом (селектор вибору цілого типу). |
| **switch (<селектор вибору>)**  **{**  **case** <знач. 1> : <оператор 1>;  **break;**  **case** < знач. 2> : <оператор 2>;  **break;**  **………………………………………….**  **case** < знач. N> : <оператор N>;  **break;**  **default :** <оператор N+1>;  **break;**  **}** | var score = parseInt(prompt("score", ""));  var result;  //-------------- Знаходимо результат за допомогою switch  switch (score)  {  case 2: result="Незадовільно";  break;  case 3: result="Задовільно.";  break;  case 4: result="Добре";  break;  case 5: result="Відмінно";  break;  default: result="Неправильна оцінка.";  break;  }  alert(result); |

**Приклад**. З клавіатури вводиться оцінка у національній шкалі, необхідно вивести повідомлення про те, чи зараховано студенту залік.

<script>

// З клавіатури ввоидться номер місяця, вивести на екран пору року

var score = parseInt(prompt("score", ""));

var result;

//-------------- Знаходимо результат за допомогою switch

switch (score)

{

case 1:

case 2: result="Незараховано";

break;

case 3:

case 4:

case 5: result="Зараховано";

break;

default: result="Неправильна оцінка.";

break;

}

//-----------------------------

alert(result);

</script>

**Приклад.**

var b= 1;

var a = 2;

switch (a) {

case 1: document.write(a);

break;

case 3.5: document.write(a); 🡨 Значення є дійсним числом 3.5

break;

case b+1: document.write(a); 🡨 Значення обраховується «b+1»

break;

}

## Цикли

Часто постає потреба виконати один і той самий оператор декілька разів. для цього застосовують *оператори циклів*. Цикл складається із *заголовка* і *тіла*. У заголовку циклу зазначається умова завершення циклу, а тіло циклу являє собою оператор, який потрібно виконати декілька разів. Кожне виконання оператора тіла циклу називається його *ітерацією*.

У JavaScript, як і в більшості інших мов програмування, є три типи циклів: з передумовою, післяумовою та параметром.

**Оператор циклу з передумовою** while

Оператор while циклічно виконує оператор (своє тіло) до тих пір, поки умова виконується (логічний вираз приймає значення true).

|  |  |  |
| --- | --- | --- |
| **Програмна структура** | **Аналог на мові блок-схем** | **Приклад.**  Знаходити суму чисел, які вводить користувач, **поки сума є меншою за 100**. |
| while( умова )  оператор ; | **умова**  Оператор  +  – | var sum=0;  while(sum<100)  sum=sum+ +prompt(‘number=’,’0’); |

Якщо тіло циклу складається з більше ніж одного оператора, то необхідно використати складений оператор (записати ці оператори у фігурних дужках).

|  |  |  |
| --- | --- | --- |
| **Програмна структура** | **Аналог на мові блок-схем** | **Приклад.**  Знаходити суму і добуток чисел, які вводить користувач, **поки сума є меншою за 100** |
| while( умова )  **{**  оператор1 ;  . . . . . .  операторN ;  **}** | **умова**  Оператор1  +  –  ОператорN | var sum=0;  var mult=1;  while(sum<=100)  **{**  var number = +prompt(‘number=’,’0’);  sum = sum + number;  mult = mult \* number;  **}** |

Іноді цикл необхідно виконати певну кількість разів. В цьому випадку використовують додаткову змінну *лічильник*, за допомогою якої контролюють кількість ітерацій (кількість разів виконання тіла циклу).

|  |  |
| --- | --- |
| Загальна форма | Приклад. 100 разів вивести на екран слово «Мир». |
| var лічильник = 0;  while( лічильник < кількість повторень )  {  оператор ;  лічильник ++ ;  } | var **i = 0**;  while (**i < 100**)  {  document.write(‘Мир’);    **i++;**  } |

Також цикл часто використовують у випадку, коли деяка величина (*параметр*) повинна змінюватися від деякого початково до кінцевого значення з певним кроком. Зауважимо, що попередній приклад є частковим випадком, коли початкове значення параметра 0, а кінцеве дорівнює кількості виконань.

|  |  |
| --- | --- |
| Загальна форма | Приклад. Вивести на екран усі числа кратні трьом від 6 до 28. |
| var параметр = початкове значення ;  while( параметр <= кінцеве значення)  {  оператор ;  параметр = параметр + крок;  } | var **i = 6**;  while (**i <= 28**)  {  document.write( i );    **i=i+3;**  } |

*Приклад. 4.1.1*  Скласти скрипт зворотного відліку який виводить числа від 5 до 1 а потім повідомлення «БУМ!»

var i=5;

while (i>0){

alert(i--);

}

alert('БУМ!');

**Цикл з післяумовою** do-while

Оператор циклу do-while відрізняється від оператора while тим, що перевірка умови виконується не до, а після виконання тіла цикла (оператора). А тому у операторі циклу do-while тіло циклу виконається принаймні один раз. Як і в циклі з передумовою, тіло виконується поки умова вірна.

|  |  |  |
| --- | --- | --- |
| **Програмна структура** | **Аналог на мові блок-схем** | **Приклад.**  Знаходити суму чисел, які вводить користувач поки не буде введено 0. Зауважимо, що принаймні одне число треба ввести (принаймні один раз тіло циклу потрібно виконати). |
| do  {  оператор ;  }  while ( умова ); | **умова**  Оператор  +  – | var sum=0;  var number;  do  {  number= +prompt(“number=”,“0”);  sum=sum+number;  }  while(number!=0); |

**Цикл з параметром for**

Найчастіше у випадку, коли повторення потрібно робити у залежності від зміни деякого параметра або ж у випадку, коли потрібно повторювати тіло циклу деяку кількість разів використовують цикл з параметром for.

Загальний вигляд:

for (<вираз ініціалізації>;<умова продовження>;<ітераційна частина>)

<оператор>;

Аналог на мові блок-схем:

Початкова ініціалізація

Умова продовження циклу

Оператор

Ітераційна частина

+

-

Схематичне зображення виконання оператора:

for(<вираз ініціалізації>;<умова продовження>;<ітераційна частина>)

<оператор>;

**Початок**

Оператор for працює у відповідності до наступного алгоритму:

1. Обчислюється вираз ініціалізації. У цій частині допустима ініціалізація декількох параметрів.
2. Перевіряється умова продовження. Якщо умова невірна, то робота циклу завершується і передається управління наступному оператору.
3. Якщо умова істинна, виконується тіло даного оператора.
4. Виконується приріст одного або декількох параметрів циклу (або виконується довільна інша операція).
5. Здійснюється перехід до кроку 2.

Розв’яжемо раніше розглянуті приклад з використанням циклу for

**Приклад 1.**

|  |  |
| --- | --- |
| Загальна форма | for( лічильник = 0; лічильник < кількість повторень ; лічильник ++)  {  оператор ;  } |
| Приклад, розв’язаний з використанням  for | //----- **100 разів** вивести на екран слово «Мир» ----  for( var **i = 0** ; **i < 100** ; **i++ )**  {  document.write(‘Мир’);  } |
| Розв’язаний з використанням  циклу while | var **i = 0**;  while (**i < 100**)  {  document.write(‘Мир’);  **i++;**  } |

Приклад 2.

|  |  |
| --- | --- |
| Загальна форма | for(**параметр=поч.значення; параметр<=кін.значення; параметр= параметр+крок** )  {  оператор ;  } |
| Приклад, розв’язаний з використанням  for | //------ Вивести на екран усі числа кратні трьом **від 6 до 28**.  for(var **i = 6**; **i <= 28**; **i=i+3 )**  {  document.write( i );  } |
| Розв’язаний з використанням  циклу while | var **i = 6**;  while (**i <= 28**)  {  document.write( i );  **i=i+3;**  } |

Як бачимо, цикл for організовано за принципом циклу while. Тільки запис є значно коротшим (і ініціалізація, і умова продовження, і зміна параметра у одному рядку).

Будь-яка частина for може бути пропущена. Наприклад, можна прибрати початкове значення. Можна прибрати і крок:

for (;умова;) {

//цикл перетворився на аналог while (умова)

}

А можна і взагалі прибрати все, отримавши нескінченний цикл:

for (;;)  // Буде виконуватися вічно

При цьому самі крапки з комою «;» обов'язково повинні бути присутніми, інакше буде помилка синтаксису.

**Опратори break і continue**

Вийти з циклу можна не тільки при перевірці умови але й, взагалі, в будь-який момент. Цю можливість забезпечує оператор break.

Модифікуємо скипт 4.2 так, щоб при натисканні відміни в користувача перепитувало чи він справді хоче вийти.

do {

var a=Math.floor(Math.random()\*100),

b=+(Math.random()\*100).toFixed(0);

var res=prompt(a+'+'+b+'=','');

if (res==null){

if (confirm('Ви справді бажаєте вийти?')){

break};

}

} while ((a+b!= +res));

if (res!=null){ alert('Вірно')};

Буває потрібно вийти одночасно з декількох рівнів циклу. Звичний виклик break не може перервати два цикли відразу. Для цього існують мітки.

Мітка має вигляд ім'я:. ім'я повинне бути унікальним. Вона кладеться перед циклом.:

exit:

for (var i = 0; i < 3; i++) {

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

var input = prompt(a[ '+i+','+j+’]’, '');  
if (input == null) break exit; // (\*)

}  
 };  
 alert('Готово!');

|  |  |
| --- | --- |
| Загальний формат опису | Приклад |
| **СПОСОБИ ОПИСУ** |  |
| **Function Declaration**  (через параметри передаємо у функцію величини, які необхідні для роботи функції)  **function** назва(параметр1, параметр2, … , параметрN ) {   тіло функції;  **return** результат;  }  Функція будується на першому етапі роботи інтерпретатора. Може бути використана іодразу | let a = sum(23, 9);  function sum(a, b) {      return a + b;  } |
| **Function Expression**  **var змінна = function** (параметр1, параметр2, … , параметрN ) {   тіло функції;  **return** результат;  }  Функція будується і присвоюється у змінну на другому етапі (виконання операторів) | var sum = function (a, b) {        return a + b;  }  let a = sum(23, 9); |
| **Лямбда вирази**  //------------ один параметр, один операнд в тілі функції (результат знаходиться одразу)  параметр => результат;  -------------------------------------------------------------------------------------------------------  //------------- багато параметрів, один операнд в тілі функції  (параметр1, параметр2, …, параметрN) => результат;  --------------------------------------------------------------------------------------------------------  //більше одного параметра у тілі функції  (параметр1, параметр2, …, параметрN) => {return результат;}    //Без параметрів  ( ) => операнд | var f = x => x \* x;          alert(f(5)); //25  ------------------------          var sum = (a, b) => a + b;          alert(sum(3,5));   //8  ------------------------  var sum = (a, b)=> {              return a + b;          }  ---------------          "use strict"          var area = (a, b, c) => {              var p = (a + b + c) / 2;              return Math.sqrt(p \* (p - a) \* (p - b) \* (p - c));          }          alert(area(3,5,4)); |
| Використання return без значення-результату дозволяє перервати роботу функції (по аналогії з break у циклах) | //До першого нуля елементи масиву множити на 2          function func(arr) {              for (var i = 0; i < arr.length; i++) {                  if (arr[i] == 0) return; //Якщо 0 то припинити обробку                  arr[i] \*= 2;              }          } |
| При використанні способу опису Function Declaration  і режиму  “use strict” функцію, яка описана в середині якогось блоку, можна використовувати тільки в середині цього ж блоку!!! | 'use strict';    if (true) {      sayHi(); // ок      function sayHi() {      alert("Привет!");    }    }  sayHi(); // Error |
| Під час виклику функції значення копіюються у локальні змінні - параметри | var a = 7;  function func(c) {  // для с створено нову пам’ять      c++;      alert("c="+c);  // c=8  }  func(a);   //значення а скопіюється у с  alert("a="+a); // a=7 (значення не змінилося) |
| Якщо функція не повертає результат через return або ж використовується return без значення-результату, то її значення – undefined | function **func**(a) {    a++;  // жодного сенсу  }    alert(func(2)); //undefined |
| Описані у функції змінні є локальними стосовно даної функції ; | function func() {              var a = 9;          }          alert(a);   //a is not defined |
| Допустиме (але небажане, окрім випадків замикання) використання зовнішніх (глобальних, описаних поза функцією) змінних | var a = 7;  function func() {      a = 9;  //**зверання до зовнішньої змінної**      alert(a);  }  func(); |
| ------------------------------------------- |  |
| **Задання значень формальних параметрів за замовчуванням** |  |
| Старий стиль (перевірка на undefined) | function func(a, b) {     if (a === undefined)         a = 10;     b = b || 7; //Обережно з значенням 0!!!     return a + b;  }  var s=func();   //значення a=10, b=7  alert("s="+s); // s=17  s = func(5); //значення a=5, b=7  alert("s=" + s); // s=12  s = func(5,4); //значення a=5, b=4  alert("s=" + s); // s=9 |
| Новий стиль (вказуємо при описі формальних параметрів)  **function** назва(параметр1=**поч.знач.1**, параметр2=**поч.знач.2**, … , параметрN=**поч.знач.N** ) {   тіло функції;  **return** результат;  } | "use strict"  function func(**a=10, b=7**) {     return a + b;  }  var s=func();   //значення a=10, b=7  alert("s="+s); // s=17  s = func(5); //значення a=5, b=7  alert("s=" + s); // s=12  s = func(5,4); //значення a=5, b=4  alert("s=" + s); // s=9 |
| Передача довільної кількості параметрів (усі значення, які передаємо у функцію заносяться у псевдо-масив **arguments**) | // Функцію знаходження довільної кількості чисел ------          function sum() {              var s = 0;              for (var i = 0; i < arguments.length; i++) {                  s += arguments[i];              }              return s;          }          var r = sum(2, 34, 23, 45);          alert(r);  //--------------------------------------------------  // Знайти розмір заробітної плати (ставка + премія - податок)  function getFinalSalary() {     //arguments[0] - ставка, arguments[1] - премія, arguments[2] - податок      return (arguments[0] + arguments[1]) \* (1 - arguments[2])  }  // Знайти розмір заробітної плати (ставка + премія - податок)  // (значення упаковуємо в об’єкт)  function getFinalSalary(data) {     return (data.salary + data.extra) \* (1 - data.tax/100);  }  alert(getFinalSalary({ salary: 9000, extra: 1000, tax: 10 })); |
| Оператор spread «…»  Дозволяє деякі значення передавати в іменовані параметри, а усі інші у змінну-масив  **function** назва(параметр1, параметр2, … , параметрN,  . . . акумулючий масив ) {   тіло функції;  **return** результат;  }  Може бути тільки один і обов'язково повинен знаходитись у кінці | // Визначити скільки елементів належать заданому діапазону          function getCount(min,max,**...numbers**) {              var count = 0;              for (var i = 0; i < numbers.length; i++) {                  if (numbers[i] >= min && numbers[i]<=max) {                      count++;                  }              }              return count;          }  alert(getCount ( 1,    10,     3, 21, 45, 3, 90, 7));               //min=1, max=10, numbers=[3, 21, 45, 3, 90, 7] |
| Деструктуризація масиву  Якщо при описі функції вказано параметри, а значення, які необхідно передати у функцію знаходяться у масиві, то ми можемо розбити масив на окремі складові з використанням  синтаксису  **функція ( . . . масив)** | "use strict"          function f(a, b, c) {              return a + b + c;          }          var m = [2, 1, 5];          alert(**f(...m)**);  //a=2, b=1, c=5 |
| Деструктуризація об’єкту | 'use strict';    let options = {    title: "Меню",    width: 100,    height: 200  };    function showMenu({title, width, height}) {    alert(title + ' ' + width + ' ' + height); // Меню 100 200  }    showMenu(options); |
| Повертаємо декілька значень з функції  За допомогою масиву  За допомогою об’єкта | function getData(a, b) {              return [a + b , a \* b]          }          let results = getData(2, 3);          alert("Sum=" + results[0] + " Prod=" + results[1]);          'use strict';          function getData(a, b) {              return { sum:a + b, prod: a \* b };          }          let results = getData(2, 3);          alert("Sum=" + results.sum + " Prod=" + results.prod); |

|  |  |  |
| --- | --- | --- |
| Зміст | Загальний вигляд | Приклад |
| створення порожнього масиву | назва\_масиву = [ ]  назва\_масиву = new Array(); | var arr = []; //Порожній масив  var t = new Array();  //Порожній масив |
| створення за допомогою літерала (вказуємо список значень) | назва\_масиву = [ список\_значень ] | var fruits = ["Яблуко", "Апельсин", "Слива"]; |
| за допомогою конструктора (створюється масив, усі елементи якого спочатку undefined) | назва = new Array( кількість\_елементів ) | arr=new Array(3); // arr = [,,] |
| за допомогою конструктора (вказуємо значення усіх елементів) | назва = new Array( список значень ) | var arr = new Array(2,3); // еквівалентно arr =  [2, 3] |

Визначення чи є масивом

|  |  |  |
| --- | --- | --- |
| Зміст | Загальний вигляд | Приклад |
| дозволяє з’ясувати, чи є змінна масивом | Array.isArray( об’єкт\_масиву ) | if ( Array.isArray(fruits ) )      alert(“fruits  є масивом”); |

Властивості масиву

|  |  |  |
| --- | --- | --- |
| Зміст | Загальний вигляд | Приклад |
| кількість елементів | назва\_масиву  . **length** | var fruits = ["Яблуко", "Апельсин", "Слива"];  alert (fruits.**length**); //3 |
|  |  |  |
|  |  |  |

Доступ до елементів масиву

|  |  |  |
| --- | --- | --- |
| Зміст | Загальний вигляд | Приклад |
| зчитування значення  (у квадратних дужках необхідно вказати індекс у квадратних дужках) | змінна =  назва\_масиву  [ **індекс** ] | var fruits = ["Яблуко", "Апельсин", "Слива"];  alert (fruits **[0]**); // Яблуко  alert (fruits **[2]**); // Слива |
| зміна значення | назва\_масиву  [ **індекс** ] = значення | var fruits = ["Яблуко", "Апельсин", "Слива"];  fruits **[0]**= “Ківі”**;**  //fruits = ["**Ківі**", "Апельсин", "Слива"]; |
|  |  |  |

Додавання елементів

|  |  |  |
| --- | --- | --- |
| Зміст | Загальний вигляд | Приклад |
| додавання в кінець |  | var newLength = fruits.push('Апельсин'); |
| додавання на початок |  | var newLength = fruits.unshift('Клубника') |
| вставка в середину |  | var removedItems = fruits.splice(pos, 0, “Груша”); |

Видалення елементів

|  |  |  |
| --- | --- | --- |
| Зміст | Загальний вигляд | Приклад |
| видалення з кінця | видалений\_елемент =  назва\_масиву . pop() | var last = fruits.pop(); |
| видалення з початку | видалений\_елемент =  назва\_масиву . shift() | var first = fruits.shift(); |
| видалення одного елемента у вказаній позиції | видалений\_елемент =  назва\_масиву . splice(pos, 1)  pos - позиція елемента, який треба видалити | var removedItem = fruits.splice(pos, 1) |
| видалення декількох елементів починаючи з вказаної позиції | видалені\_елементи =  назва\_масиву . splice(pos, n)  pos – початкова позиція, n – кількість елементів для видалення | var removedItems = vegetables.splice(pos, n); |
| видалення існуючих і вставка нових елементів | arr.splice(pos[, deleteCount, elem1, ..., elemN])  pos – позиція, з якої починається видалення  deleteCount – кількість елементів, які потрібно видалити  elem1, ..., elemN – список елементів, які потрібно вставити | var arr = ["Я", "зараз", "вивчаю", "JavaScript"];  // Видалитит 3 перших елементи і замість них додати слова  :  "Ми", "вивчаємо"  arr.splice(0, 3, "Ми", "вивчаємо")  alert( arr ) // ["Ми", "вивчаємо", "JavaScript"] |
| видалення значення елемента (сам елемент залишається і дорівнює undefined) | delete назва\_масиву [ індекс ] | var arr = ["Я", "йду", "додому"];  **delete** arr[1]; // видалити значення з інексом 1  alert( arr); // arr = ["Я", **undefined**, "додому"]; |
| використання властивості  length  (якщо властивості length присвоїти значення, яке є меншим за справжню довжину масиву, то масив зменшується до вказаної кількості елементів) | назва\_масиву . length = нова\_довжина\_масиву | var arr = ["Я", "зараз", "вивчаю", "JavaScript"];  arr.**length** = 2; //довжина=4, а ми присвоюємо 2  alert( arr ) // ["Я", "зараз"] |

Пошук елементів

|  |  |  |
| --- | --- | --- |
| Зміст | Загальний вигляд | Приклад |
| знаходження індексу першого входження елемента (повертає -1, якщо немає елемента) | назва\_масиву . indexOf() | var fruits = ["Яблуко", "Апельсин", "Слива"];  var pos = fruits.indexOf("Слива”);  alert(pos); // 2  pos = fruits.indexOf('Банан');  alert(pos); // -1 (елементу немає в масиві)  //-------------                     //0  1  2  3  4  5  6  7  8  9  10 11 12              var a = [2, 7, 6, 9, 4, 3, 4, 6, 2, 3, 9, 6, 7]              var r = a.indexOf(3); //5              var r = a.indexOf(6); //2              var r = a.lastIndexOf(3); //9              var r = a.lastIndexOf(6); //11 |
| знаходження індексу останнього входження (пошук з кінця) | назва\_масиву . lastIndexOf () | var a=[ 11, 55,23, 11, 90]  var pos = a. lastIndexOf (11);  alert(a); // 3     (індекс останнього входження 11) |

Копіювання частини елементів

|  |  |  |
| --- | --- | --- |
| Зміст | Загальний вигляд | Приклад |
| копіювання всього масиву | назва\_масиву . slice () | var shallowCopy = fruits.slice(); |
| копіювання частини масиву | назва\_масиву . slice ( start, end )  start – індекс початку копіювання  end – індекс кінцевого елементу (до якого елемента буде здійснюватися копіювання; сам елемент не входить до скопійованої частини) | var a = [25, 12, 18, 46, 35, 26];  var t = a.slice(2, 4);  alert(t); // [18, 46] |

Інші методи

|  |  |  |
| --- | --- | --- |
| Зміст | Загальний вигляд | Приклад |
| обертання масиву (елементи будуть розміщені у зворотному порядку) | назва\_масиву .reverse() | var a = [25, 12, 18, 46, 35, 26];  a.reverse();  document.write(a); // [26,35,46,18,12,25] |
| об’єднання масиву  (повертає **новий масив**, який складається з елементів даного масиву і масиву, з яким виконується об’єднання) | масив1 . concat(масив2 )    масив . concat(список\_нових\_масивів )    масив . concat( список\_нових\_елементів ) | var a1 = [1, 2, 3];          var a2 = [4, 5];          var a3 = a1.concat(a2);          document.write(a3); // 1,2,3,4,5  //--------------------------          var a1 = [1, 2, 3];          var a2 = [4, 5];          var a3 = [6,7];          var a4 = a1.concat(a2,a3);          document.write(a4); // 1,2,3,4,5,6,7  //--------------------------          var a1 = [1, 2, 3];          var a2 = a1.concat(4,5);          document.write(a2); // 1,2,3,4,5 |
| створення рядка з елементів масиву  (елементи розділяються вказаним знаком-розділювачем) | масив . join( символ-розділювач )    масив . join()    //символом розділювачем є кома | var arr = ['Марія', 'Петро', 'Марина', 'Василь'];  var str = arr.**join(';')**;  alert( str ); // "Марія;Петро;Марина;Василь"  //-------------------------------------  var arr = ['Марія', 'Петро', 'Марина', 'Василь'];  var str = arr.join();  alert(str); // Марія,Петро,Марина,Василь |
| створення масиву шляхом розбиття рядка, у якому елементи розділені деяким символом-розділювачем |  | var arr=” Маша;Петя;Марина;Василий”.split(“;”) |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Призначення** | **Що повертає** | **Особливості функції-колбека** | **Приклад** |
| **sort** | упорядкування елементів за вказаним правилом | змінюється сам масив | function compareRule(item1, item2)  {     ...  }  Функція-колбек задає правило попарного порівняння елементів під час сортування і повинна повертати:   * число більше за нуль, якщо item1>item2; * число менше за нуль, якщо item1<item2; * нуль, якщо item1 = item2; | //--------- без правила порівняння ---  var a = [12, 3, -21, 4, -56, 4, 3];  a.sort(); //елементи порівнюються як string  alert(a); // -21,-56,12,3,3,4,4  //--задаємо правило порівняння (за зростанням)-  function rule(item1, item2) {      if (item1 > item2)          return 1;      else          if (item1 < item2)             return -1;          else              return 0;  }  var a = [12, 3, -21, 4, -56, 4, 3];  a.sort(rule);  alert(a); // -56,-21,3,3,4,4,12  //-- З використанням лямбда-виразів ---  var a = [12, 3, -21, 4, -56, 4, 3];  a.sort((a, b) => a - b);  alert(a); |
| **ForEach** | Коли треба переглянути елементи не змінюючи їх, або треба змінити елементи існуючого масиву (новий масив не формуємо). Елементи перебираються по черзі зліва-направо. | Нічого не повертає | function callback(itemCopy, index, array)  { . . .}  itemCopy – копія поточного елемента  масиву (окрема змінна); index – номер поточного елемента; array – посилання на масив, для якого викликано метод.  Функція-колбек нічого не повертає.  Якщо хочемо змінювати елементи використовуємо посилання на масив (array[index]= нове\_значення) | Вивести парні елементи.  var a = [1, 2, 3, 4];  function isEven(x, i, arr) {      if (x % 2 == 0)        alert(x);  }  a**.forEach(isEven);**  Усі елементи помножити на 2  var a = [1, 2, 3];  a.**forEach((x, i, arr) => { arr[i] \*= 2;})**  alert(arr); // [2,4,6] |
| **Map** | Коли треба сформувати новий масив на основі існуючого (кількість елементів нового масиву співпадає з кількістю існуючого) | Повертає новий масив, який сформовано на основі даного (правило утворення елементів нового масиву задає функція-колбек) | function callback(itemCopy, index, array)  {     . . .  return **обчислене\_значення\_елемента\_нового\_масиву**  }  Функція-колбек задає правило утворення елементів нового масиву і повертає обчислене значення елемента нового масиву на основі поточного значення itemCopy початкового масиву | Створити новий масив, у якому до кожного елемента додано 20  function rule(x, i, arr) {     return x + 20;  }  var a = [12, 3, -21, 4, -56, 4, 3];  var newArr=a.map(rule);  alert(newArr); //32,23,-1,24,-36,24,23  Створити новий масив, у якому усі елементи помножено на 7  var a = [1, 2, 3];  var newArr=a.map((x)=>x\*7);  alert(newArr); //7,14,21 |
| **Filter** | Коли треба сформувати новий масив з елементів даного масиву, які задовольняють деякій умові (тобто елементи існуючого або входять у новий масив, або не входять) | Повертає масив елементів, які задовольняють заданій умові | function callback(itemCopy, index, array)  {     . . .  return **результат\_перевірки\_умови\_фільтрації**  }  Фунція-кобек повертає True (коли елемент підходить за фільтром) або False (коли елемент не підходить за фільтром) | Дано масив цін. Сформувати масив цін, які більші за 100  function rule(x, i, arr) {      return x > 100;  }  var a = [450, 13, 23, 900];  var newArr = a.filter(rule);  alert(newArr); // 450,900  //-------------------------------------  Дано масив років народження працівників. Вивести тих, які народилися між 1980 і 2003 роками.  var a = [1990, 2017, 2006, 2001];  var newArr = a.filter((x)=> x>=1980&&x<=2003);  alert(newArr); // 1990,2001 |
| **Every** | Коли треба перевірити чи для усіх елементів масиву виконується деяка умова (яку записуємо у функції) | every повертає true, якщо усі елементи задовольняють заданій умові | function callback(itemCopy, index, array)  {     . . .  return **результат\_перевірки\_умови**  }  Фунція-кобек повертає True  або False (у функції повертаємо результат перевірки умови) | Дано масив оцінок учня. З’ясувати, чи є учень двійочником.  function isNotBad(mark) { // чи не двійка      return mark > 3;  }  var marks = [8, 11, 9, 10, 9];  if (marks.**every(isNotBad)**)      alert("Не двійочник");  else      alert("Двійочник"); |
| **Some** | Коли треба перевірити чи хоча би для одного з елементів виконується умова (яку записуємо у функції) | some повертає true, якщо хоча би один елемент задовольняє заданій умові | function callback(itemCopy, index, array)  {     . . .  return **результат\_перевірки\_умови**  }  Фунція-кобек повертає True  або False (у функції повертаємо результат перевірки умови) | Дано масив оцінок учня. З’ясувати, чи є у нього оцінки 10.  var marks = [8, 10, 9, 11, 9];  if (marks.**some(mark=>mark==10)**)      alert("Є оцінка 10");  else      alert("Немає оцінки 10"); |
| **reduce** | reduce(callback [, initialValue])  послідовно викликає функцію callback один раз для кожного елемента масиву (за виключенням порожніх елементів undefined), обчислюючи при цьому деяку величину. Значення величини при цьому поступово (зліва-направо) обраховується (накопичується) на основі кожного елемента масиву. Наприклад, при знаходженні суми до величини суми поступово додається кожен елемент масиву. | повертає останнє обчислене значення акумульованої величини | function callback(**previousValue**,itemCopy,index,array)  {    . . .  }  previousValue – результат виклику функції для попереднього елемента (проміжковий результат акумульованої величини). previousValue при першому виклику дорівнює;  initialValue. Якщо initialValue не вказано, то previousValue спочатку дорівнює першому елементу масиву, а перебір починається з другого | Знайти кількість додатних  function kilk(result, x, i, arr) {      if (x > 0) { result++ };      return result;  }  var a = [1, 2, -3, 4, -5];  alert(a.**reduce**(kilk, **0**); //спочатку result=0  //-------------------------------  Знайти суму елементів масиву  function suma(result, x, i, arr) {      result += x;      return result;  }  var a = [1, 2, -3, 4, -5];  alert(a.**reduce**(suma));  //спочатку result=a[0] |
| **reduceRight** | reduceRight( callback [, initialValue])  послідовно викликає функцію callback один раз для кожного елемента масиву (за виключенням порожніх елементів undefined), обчислюючи при цьому деяку величину. Значення величини при цьому поступово (зліва-направо) обраховується (накопичується) на основі кожного елемента масиву. Наприклад, при знаходженні суми до величини суми поступово додається кожен елемент масиву. | повертає останнє обчислене значення акумульованої величини | function callback(**previousValue**,itemCopy,index,array)  {    . . .  }  previousValue – результат виклику функції для попереднього елемента (проміжковий результат акумульованої величини). previousValue при першому виклику дорівнює;  initialValue. Якщо initialValue не вказано, то previousValue спочатку дорівнює першому елементу масиву, а перебір починається з другого | Початкове значення суми дорівнює останньому елементу. Поступово розглядаючи елементи справа-наліво додаємо наступний елемент тільки у тому випадку, якщо його значення менше за поточне значення суми.  var a = [1, 2, 3];  alert(a.reduceRight(     (currentResult, x) =>               x < currentResult ?               currentResult + x :               currentResult)  );  //6 |