На сьогодні, використовуючи JavaScript, можна створювати:

* веб-застосунки на фреймворках React, Vue, Angular та інших
* бекенд-застосунки на Node.js
* мобільні застосунки на React Native або Ionic
* десктоп-застосунки за допомогою Electron
* мікроконтролери з Johnny-Five і Espruino

Для реалізації всіх цих можливостей, необхідно добре знати синтаксис мови, її особливості та механізми, тренуватися мислити алгоритмічно і вирішувати якомога більше практичних задач.

**Підключення скрипту**

Щоб додати скрипт на веб-сторінку, в HTML-файлі використовується тег script, в атрибуті src якого, зазначаємо посилання на зовнішній JavaScript-файл.

Щоб підключити JavaScript із зовнішнього файлу:

* Створіть файл з розширенням .js і помістіть його у підпапку js.
* Потім вкажіть шлях до файлу скрипту в атрибуті src тегу script.

*<!DOCTYPE html>*  
<html lang="en">  
 <head>  
 <meta charset="UTF-8" />  
 <title>JavaScript is fun!</title>  
 </head>  
  
 <body>  
 *<!-- контент -->*  
 <script src="js/script.js"></script>  
 </body>  
</html>

ЦІКАВО

Розміщення JavaScript файлу в папці js не вимагається, однак, - це хороша практика.

Якщо скрипт підключений в <head>, рендеринг сторінки зупиняється доти, доки скрипт не завантажиться і повністю виконається. Браузер завантажує і відображає HTML поступово. Якщо він бачить тег <script>, без додаткових атрибутів, то спочатку виконується скрипт, і тільки потім обробляється інший код HTML-файлу. Тому, скрипт підключають перед закриваючим тегом <body>, після всього вмісту, як у прикладі.

# Інструменти розробника

При написанні коду завжди будуть сповіщення і помилки, - це нормально. Консоль браузера показує інформацію, пов'язану із веб-сторінкою, зокрема повідомлення про помилки і попередження виконання JS-коду, запущеного в контексті сторінки.

Наступні комбінації клавіш відкривають інструменти розробника на вкладці Console:

* Windows і Linux - Ctrl + Shift + J або F12
* MacOS - Command + Option + J

# Основи синтаксису

При написанні коду важливо не просто знати, який символ або конструкцію можна використовувати, але насамперед необхідно розуміти термінологію і складові вихідного коду. В цій секції нам не важливе розуміння як що працює, ми тільки познайомимось з базовою термінологією і синтаксисом.

## Інструкція[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/syntax#%D1%96%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%86%D1%96%D1%8F)

**Інструкція (statement)** - це пов'язаний набір слів і символів із синтаксису мови, які об'єднуються з метою вираження однієї ідеї, однієї інструкції для машини.

a = b \* 2;

Інструкції в JavaScript завершуються крапкою з комою, яку можна порівняти із крапкою наприкінці речення вашої рідної мови.

* a і b - змінні (як в алгебраїчному рівнянні), це сховища даних, які використовує програма. Змінна складається із ідентифікатора (імені) і пов'язаного з ним значення.
* 2 - просто число. Це називається значенням літерала (literal value), тому що не зберігається у змінній.
* = і \* - оператори, вчиняють дії над значеннями і змінними.

Уявімо, що змінна b вже зберігає число 10. Тоді ця інструкція каже машині:

1. Піди знайди змінну з ідентифікатором b і запитай, яке в неї зараз значення.
2. Підстав значення змінної b (10), у твердження на місце b.
3. Виконай операцію множення 10 на 2.
4. Запиши результат обчислення виразу правої частини у змінну a.

ЦІКАВО

Завершення інструкції крапкою з комою не вимагається, однак, **настійно рекомендується** завжди її ставити. Це просте правило зробить код зрозумілішим і допоможе уникнути неочевидних помилок.

## Вираз[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/syntax#%D0%B2%D0%B8%D1%80%D0%B0%D0%B7)

Інструкції складаються із частин, як і в будь-якій мові речення складаються із фраз і ці фрази називаються виразами.

**Вираз (expression)** - посилання на змінну або значення, або на набір змінних і значень в поєднанні з операторами.

[ [a] = [ [b] \* [2] ] ]

Інструкція з прикладу вище містить 5 виразів, які виділені квадратними дужками для візуалізації (це не синтаксис мови):

* [2] - вираз значення літерала.
* [b]і[a] - вирази змінної, означають необхідність підставити значення змінної, але лише у тому випадку, якщо змінна складається із правої частини виразу присвоювання.
* [b \* 2] - арифметичний вираз множення.
* [a = b \* 2] - вираз присвоювання. У нашому випадку вказує на необхідність обчислення правої частини виразу і присвоювання результату змінної a у лівій частині виразу.

Також існує вираз виклику, порівняння тощо. Ми не будемо зараз розглядати їх усіх, нам важливо розуміти, з яких частин складається вихідний код і як правильно його читати.

## Інтерфейс[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/syntax#%D1%96%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81)

Коли ми підходимо до автомату з кавою або сідаємо за кермо автомобіля, існує певний набір елементів управління, з яким можна взаємодіяти. У програмуванні це називається інтерфейс.

**Інтерфейс** - це набір властивостей і методів сутності, доступних для використання у вихідному коді.

### Властивість[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/syntax#%D0%B2%D0%BB%D0%B0%D1%81%D1%82%D0%B8%D0%B2%D1%96%D1%81%D1%82%D1%8C)

У нас з вами є властивості: зріст, вага, колір очей, тобто якісь описові характеристики. Так само і у даних є властивості, наприклад у рядка є властивість його довжини. Синтаксис звернення до властивості дуже простий - через крапку.

сутність.ім`я\_властивості

Для наочності, звернемось до властивості рядка length, яка містить кіл-ть символів рядка.

"JavaScript is awesome".length;

### Метод[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/syntax#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4)

Це виклик дії, наприклад присісти або плавати, тобто якась активна операція. Так само і у даних є свої заздалегідь визначені методи, наприклад, можна додати або видалити елементи з колекції, перевести рядок в інший регістр тощо. Синтаксис виклику метода дуже схожий на звернення до властивості, але в кінці додається пара круглих дужок.

сутність.ім`я\_метода()

Для прикладу звернемось до методу рядка toUpperCase(), який зробить усі літери великими.

"JavaScript is awesome".toUpperCase();

## Суворий режим[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/syntax#%D1%81%D1%83%D0%B2%D0%BE%D1%80%D0%B8%D0%B9-%D1%80%D0%B5%D0%B6%D0%B8%D0%BC)

Нова можливість у специфікації ECMAScript 5, яка дозволяє переводити скрипт у режим повної відповідності сучасному стандарту. Це запобігає певним помилкам, як-от використання небезпечних і застарілих конструкцій.

Для того, щоб перевести скрипт в суворий режим, достатньо зазначити директиву на початку js-файлу. Завжди пишіть код в суворому режимі.

script.js

"use strict";  
*// Це коментар. Далі йде увесь код JS-файлу*

# Змінні і типи

Змінні використовуються для зберігання даних і складаються із ідентифікатора (імені) і області в пам'яті, де зберігається їх значення. Змінну можна уявити у вигляді коробки з назвою, в якій щось лежить (значення).

<ключове\_слово> <ім`я\_змінної> = <значення>

## Імена змінних[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/variables#%D1%96%D0%BC%D0%B5%D0%BD%D0%B0-%D0%B7%D0%BC%D1%96%D0%BD%D0%BD%D0%B8%D1%85)

**Ідентифікатор** - це ім'я змінної, функції або класу. Складається з одного або декількох символів у наступному форматі.

* Першим символом повинна бути літера a-z або A-Z, символ підкреслення \_ або знак долара $.
* Інші символи можуть бути літерами a-z, A-Z, цифрами 0-9, підкресленнями \_ і знаками долара $.
* Ідентифікатори чуттєві до регістру. Це означає, що змінні user, usEr і User - різні.

Ім'я змінної повинно бути зрозумілим.

*# ❌ Погано*  
chislo  
korzina\_tovarov  
profil\_polzovatelya  
tekushiy\_mesyaz  
  
*# ✅ Добре*  
number  
cart  
userProfile  
currentMonth

Використання camelCase-нотації для ідентифікаторів - це стандарт. Перше слово пишеться малими літерами, а кожне наступне починається з великої. Наприклад, user, greetUser, getUserData, isActive, activeGuestCount, totalWorkerSalary.

* [Як називати змінні](https://bookflow.ru/kak-nazyvat-peremennye/)
* [Домовленість про імена змінних](https://medium.freecodecamp.org/javascript-naming-conventions-dos-and-don-ts-99c0e2fdd78a)

## Ключові слова[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/variables#%D0%BA%D0%BB%D1%8E%D1%87%D0%BE%D0%B2%D1%96-%D1%81%D0%BB%D0%BE%D0%B2%D0%B0)

Існує список зарезервованих ключових слів, які мають спеціальне значення і використовуються для певних конструкцій. Забороняється використовувати ключові слова у якості ідентифікаторів.

|  |  |  |  |
| --- | --- | --- | --- |
| abstract | arguments | await | boolean |
| break | byte | case | catch |
| char | class | const | continue |
| debugger | default | delete | do |
| double | else | enum | eval |
| export | extends | false | final |
| finally | float | for | function |
| goto | if | implements | import |
| in | instanceof | int | interface |
| let | long | native | new |
| null | package | private | protected |
| public | return | short | static |
| super | switch | synchronized | this |
| throw | throws | transient | true |
| try | typeof | var | void |
| volatile | while | with | yield |

## Оголошення змінних[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/variables#%D0%BE%D0%B3%D0%BE%D0%BB%D0%BE%D1%88%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B7%D0%BC%D1%96%D0%BD%D0%BD%D0%B8%D1%85)

Оголошення змінної починається з ключового слова const. Така змінна повинна бути одразу ініціалізована значенням, після чого її не можна перевизначити.

*// Змінні, оголошені як const, обов'язково повинні бути ініціалізовані*  
*// значенням під час оголошення, інакше виникне помилка.*  
const yearOfBirth = 2006;  
console.log(yearOfBirth); *// 2006*  
  
*// Якщо зміна оголошена як const, перезаписати її значення не можна.*  
*// При спробі присвоїти нове значення, виникне помилка виконання скрипту.*  
yearOfBirth = 2020; *// ❌ Неправильно, виникне помилка*

Для того, щоб оголосити змінну, якій у подальшому можна буде присвоїти нове значення, використовується ключове слово let.

*// Змінним, оголошеним через let, не обов'язково одразу присвоювати значення.*  
let age;  
  
*// Якщо змінній, оголошеній як let, не було присвоєно значення,*  
*// вона ініціалізується спеціальним значенням undefined (не визначено).*  
console.log(age); *// undefined*  
  
*// console.log() - це метод для виведення даних у консоль браузера,*  
*// пізніше познайомимось з ним детальніше.*  
  
*// Якщо змінна оголошена як let, її значення можна перезаписати.*  
age = 14;  
console.log(age); *// 14*

ЦІКАВО

Оголошення змінної без ключового слова let або const призведе до помилки, якщо скрипт виконується в суворому режимі.

## Коли використовувати const і let[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/variables" \l "%D0%BA%D0%BE%D0%BB%D0%B8-%D0%B2%D0%B8%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D0%BE%D0%B2%D1%83%D0%B2%D0%B0%D1%82%D0%B8-const-%D1%96-let" \o "Direct link to heading)

Єдина відмінність const і let полягає у тому, що const забороняє повторно присвоювати змінній будь-яке значення. Оголошення const робить код читабельнішим, тому що змінна завжди посилається на одне і те саме зачення. У випадку з let такої впевненості немає.

Буде розумно використовувати let і const наступним чином:

* Використовуйте const за замовчуванням, більшість змінних будуть оголошені саме таким чином.
* Використовуйте let, якщо необхідно присвоїти змінній інше значення під час виконання скрипту.

## константи і КОНСТАНТИ[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/variables#%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D0%B0%D0%BD%D1%82%D0%B8-%D1%96-%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D0%B0%D0%BD%D1%82%D0%B8)

Імена КОНСТАНТ - змінних, значення яких не змінюється ніколи впродовж роботи всього скрипту, як правило, записуються у форматі UPPER\_SNAKE\_CASE.

*// Константа, що зберігає значення кольору*  
const COLOR\_TEAL = "#009688";  
  
*// Константа, що зберігає повідомлення про результат логіну*  
const LOGIN\_SUCCESS\_MESSAGE = "Ласкаво просимо!";

Абсолютна більшість змінних – константи в іншому сенсі, вони просто не змінюють значення після присвоєння. Але, у різних запусках скрипту, це значення може бути різним. Імена таких змінних записують за допомогою формату camelCase.

## Звернення до змінної[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/variables#%D0%B7%D0%B2%D0%B5%D1%80%D0%BD%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B4%D0%BE-%D0%B7%D0%BC%D1%96%D0%BD%D0%BD%D0%BE%D1%97)

Важливо розрізняти невизначені і неоголошені змінні.

**Невизначена (undefined)** - це змінна, яка була оголошена ключовим словом let, але не ініціалізована значенням. За замовчуванням їй присвоюється початкове значення undefined.

let username;  
console.log(username); *// undefined*

**Неоголошена (undeclared або not defined)** - це змінна, яка не була оголошена в доступній області видимості. Спроба звернутися до змінної до її оголошення призведе до помилки. Наприклад, щоб прочитати або змінити її значення.

*// ❌ Неправильно, буде помилка*  
age = 15; *// ReferenceError: Cannot access 'age' before initialization*  
console.log(age); *// ReferenceError: age is not defined*  
  
*// Оголошення змінної age*  
let age = 20;  
  
*// ✅ Правильно, звертаємось після оголошення*  
age = 25;  
console.log(age); *// 25*

## Примітивні типи[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/variables#%D0%BF%D1%80%D0%B8%D0%BC%D1%96%D1%82%D0%B8%D0%B2%D0%BD%D1%96-%D1%82%D0%B8%D0%BF%D0%B8)

В JavaScript змінна не асоціюється з будь-яким типом даних, тип має її значення. Тобто змінна може зберігати значення різних типів.

**Number** - цілі числа і числа з плаваючою комою (крапкою).

const age = 20;  
const points = 15.8;

**String** - рядки, послідовність з нуля або більше символів. Рядок починається і закінчується одинарними ', або подвійними лапками ".

const username = "Mango";  
const description = "JavaSript для початківців";

**Boolean** - логічний тип даних, прапорці стану. Всього два значення: true і false. Наприклад, на запитання чи увімкнено світло в кімнаті, можна відповісти так (true) або ні (false).

* true — так, вірно, істина, 1
* false — ні, невірно, неправда, 0

Зверніть увагу на імена змінних, що містять буль. Вони ставлять запитання, і відповідь на нього - так або ні.

const isLoggedIn = true;  
const canMerge = false;  
const hasChildren = true;  
const isModalOpen = false;

**null** - особливе значення, яке по суті означає ніщо. Використовується в тих ситуаціях, коли необхідно явно вказати порожнечу. Наприклад, якщо користувач нічого не вибрав, то можна сказати що значення null.

let selectedProduct = null;

**undefined** - ще одне спеціальне значення. За замовчуванням, коли змінна оголошується, але не ініціалізується, її значення не визначено, їй присвоюється undefined.

let username;  
console.log(username); *// undefined*

## Оператор typeof[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/variables" \l "%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80-typeof" \o "Direct link to heading)

Використовується для отримання типу значення змінної. Повертає на місце свого виклику тип значення змінної, вказаного після нього - рядок, в якому вказано тип.

let username;  
console.log(typeof username); *// "undefined"*  
  
let inputValue = null;  
console.log(typeof inputValue); *// "object"*  
  
const quantity = 17;  
console.log(typeof quantity); *// "number"*  
  
const message = "JavaScript is awesome!";  
console.log(typeof message); *// "string"*  
  
const isSidebarOpen = false;  
console.log(typeof isSidebarOpen); *// "boolean"*

# Взаємодія з користувачем

Розберемо базові операції введення/виведення, достатні для отримання і відображення даних від користувача, перш ніж навчимося працювати з HTML-документом.

## Виведення даних[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/user-input#%D0%B2%D0%B8%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B4%D0%B0%D0%BD%D0%B8%D1%85)

Для виведення даних існує два методи: console.log() і alert(). В круглих дужках зазначаємо ім'я змінної, значення якої необхідно вивести.

const message = "JavaScript is awesome!";  
console.log(message); *// JavaScript is awesome!*

Спочатку можна вказати будь-який описовий рядок, після чого поставити кому і вказати ім'я змінної.

const username = "Mango";  
console.log("Username is ", username); *// Username is Mango*

Метод alert() виводить модальне вікно, текст якого відповідає значенню змінної (або літерала), яку передаємо в дужках.

const message = "JavaScript is awesome!";  
alert(message);

ЦІКАВО

console і alert - частина інтерфейсу window - глобального об'єкта, доступного при виконанні скрипту на веб-сторінці. Запис window.alert() зайвий, пишемо просто alert() або console.log(). Детальніше про це будемо говорити далі.

[**Going beyond console.log()**](https://medium.com/free-code-camp/how-to-use-the-javascript-console-going-beyond-console-log-5128af9d573b)

## Отримання даних[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/user-input#%D0%BE%D1%82%D1%80%D0%B8%D0%BC%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B4%D0%B0%D0%BD%D0%B8%D1%85)

Це також методи з інтерфейсу window. Результатом свого виконання вони повертають те, що було введено користувачем, тому результат їх роботи можна записати у змінну для подальшого використання.

confirm() - виводить модальне вікно з повідомленням, і дві кнопки, Ok і Cancel. Натискаючи на Ok, результатом будет true, натискаючи на Cancel - повертається false.

*// Просимо клієнта підтвердити бронювання готелю*  
*// і зберігаємо результат роботи confirm у змінну*  
const isComing = confirm("Please confirm hotel reservation");  
console.log(isComing);

prompt() - виводить модальне вікно з полем введення і кнопками Ok і Cancel. Натискаючи на Ok, результатом буде те, що ввів користувач, у випадку Cancel - повертається null.

*// Запитуємо назву готеля, в якому хотів би зупинитися клієнт*  
*// і зберігаємо результат виклику prompt у змінну.*  
const hotelName = prompt("Please enter desired hotel name");  
console.log(hotelName);

Важлива особливість prompt полягає в тому, що незалежно від того, що ввів користувач, завжди повернеться рядок. Тобто, якщо користувач ввів 5, то повернеться не число 5, а рядок "5".

const value = prompt("Please enter a number!");  
console.log(typeof value); *// "string"*  
console.log(value); *// "5"*

[Previous](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/variables)

# Основні оператори

## Математичні оператори[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/operators#%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%BD%D1%96-%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%B8)

Призначення, функціонал і пріоритет (порядок) операцій нічим не відрізняються від шкільного курсу алгебри. Оператори повертають значення у вигляді результату виразу.

const x = 10;  
const y = 5;  
  
*// Додавання*  
console.log(x + y); *// 15*  
  
*// Віднімання*  
console.log(x - y); *// 5*  
  
*// Множення*  
console.log(x \* y); *// 50*  
  
*// Ділення*  
console.log(x / y); *// 2*  
  
*// Остача від ділення*  
console.log(x % y); *// 0*  
  
*// Додавання із заміною (також є для всіх інших операторів)*  
let value = 5;  
  
*// Аналогічно запису value = value + 10;*  
value += 10;  
console.log(value); *// 15*

ЦІКАВО

Важливо запам'ятати терміни складових виразу. + - \* / % називаються **оператори**, а те, на чому вони застосовуються - **операнди**.

## Оператори порівняння[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/operators#%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%B8-%D0%BF%D0%BE%D1%80%D1%96%D0%B2%D0%BD%D1%8F%D0%BD%D0%BD%D1%8F)

Використовуються для порівняння двох значень. Результатом свого виконання повертають буль - true або false, тобто «так» або «ні».

* a > b і a < b - більше/менше
* a >= b і a <= b - більше/менше або дорівнює
* a == b - рівність
* a != b - нерівність
* a === b - строга рівність
* a !== b - строга нерівність

const x = 5;  
const y = 10;  
const z = 5;  
  
console.log("x > y:", x > y); *// false*  
console.log("x < y:", x < y); *// true*  
console.log("x < z:", x < z); *// false*  
console.log("x <= z:", x <= z); *// true*  
console.log("x === y:", x === y); *// false*  
console.log("x === z:", x === z); *// true*  
console.log("x !== y:", x !== y); *// true*  
console.log("x !== z:", x !== z); *// false*

## Оператори рівності[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/operators#%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%B8-%D1%80%D1%96%D0%B2%D0%BD%D0%BE%D1%81%D1%82%D1%96)

«Нестрогі» оператори рівності == і != виконують перетворення типів порівнюваних значень у число, що може призвести до помилок, особливо у початківців.

*// ❌ Погано, виконується приведення типів*  
console.log(5 == "5"); *// true*  
console.log(5 != "5"); *// false*  
console.log(1 == true); *// true*  
console.log(1 != true); *// false*

На наступній ілюстрації показана таблиця порівняння значень з використанням операторів нестрогої рівності.

# Числа

Усі числа в JavaScript, як цілі, так і дробові, мають тип Number і їх можна записувати не тільки в десятковій системі числення.

## Приведення до числа[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/numbers#%D0%BF%D1%80%D0%B8%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B4%D0%BE-%D1%87%D0%B8%D1%81%D0%BB%D0%B0)

Більшість арифметичних операцій і математичних функцій перетворюють значення у число автоматично. Для того, щоб зробити це явно, використовуйте функцію Number(val), передаючи їй у val те, що потрібно привести до числа.

Якщо значення привести до числа неможливо, результатом буде спеціальне числове значення NaN (Not a Number). Аналогічним чином відбувається перетворення і в інших математичних операторах і функціях.

const valueA = "5";  
console.log(Number(valueA)); *// 5*  
console.log(typeof Number(valueA)); *// "number"*  
  
const valueB = "random string";  
console.log(Number(valueB)); *// NaN*  
console.log(typeof Number(valueB)); *// "number"*

## Методи Number.parseInt() і Number.parseFloat()[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/numbers#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8-numberparseint-%D1%96-numberparsefloat)

Перетворюють рядок символ за символом, доки це можливо. У разі виникнення помилки повертається підсумкове число.

Метод Number.parseInt() парсить з рядка ціле число.

console.log(Number.parseInt("5px")); *// 5*  
console.log(Number.parseInt("12qwe74")); *// 12*  
console.log(Number.parseInt("12.46qwe79")); *// 12*  
console.log(Number.parseInt("qweqwe")); *// NaN*

Метод Number.parseFloat() парсить з рядка дробове число.

console.log(Number.parseFloat("5px")); *// 5*  
console.log(Number.parseFloat("12qwe74")); *// 12*  
console.log(Number.parseFloat("12.46qwe79")); *// 12.46*  
console.log(Number.parseFloat("qweqwe")); *// NaN*

## Перевірка на число[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/numbers#%D0%BF%D0%B5%D1%80%D0%B5%D0%B2%D1%96%D1%80%D0%BA%D0%B0-%D0%BD%D0%B0-%D1%87%D0%B8%D1%81%D0%BB%D0%BE)

Для перевірки на число можна використовувати метод Number.isNaN(val). Він перевіряє, чи вказане значення є NaN. Метод відповідає на запитання "Це Not A Number?" і повертає:

* true - якщо значення val - NaN
* false - якщо значення val - не NaN

Для всіх значень val, крім NaN, при передачі в Number.isNaN(val) поверне false. Цей метод не намагається перетворити val в число, а просто виконує перевірку на NaN.

const validNumber = Number("51"); *// 51*  
console.log(Number.isNaN(validNumber)); *// false*  
  
const invalidNumber = Number("qweqwe"); *// NaN*  
console.log(Number.isNaN(invalidNumber)); *// true*

## Додавання чисел з рухомою крапкою (комою)[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/numbers#%D0%B4%D0%BE%D0%B4%D0%B0%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F-%D1%87%D0%B8%D1%81%D0%B5%D0%BB-%D0%B7-%D1%80%D1%83%D1%85%D0%BE%D0%BC%D0%BE%D1%8E-%D0%BA%D1%80%D0%B0%D0%BF%D0%BA%D0%BE%D1%8E-%D0%BA%D0%BE%D0%BC%D0%BE%D1%8E)

Додаючи нецілі числа, в JavaScript і в інших мовах програмування є особливість. Якщо коротко, то 0.1 + 0.2 не дорівнює 0.3, результат додавання буде більший, ніж 0.3. Це тому що машина рахує у двійковій системі.

Число 0.1 у двійковій системі числення - це нескінченний дріб, оскільки у двійковій системі одиниця не ділиться на десять. Двійкове значення нескінченних дробів зберігається тільки до певного знаку, тому виникає неточність. При додаванні 0.1 і 0.2, додаються дві неточності, виходить незначна, але все-таки помилка в обчисленнях.

console.log(0.1 + 0.2 === 0.3); *// false*  
console.log(0.1 + 0.2); *// 0.30000000000000004*

Звичайно, це не значить, що точні обчислення для таких чисел неможливі. Існує декілька методів вирішення цієї проблеми.

Можна зробити їх цілими, помноживши на N, додати, а потім результат поділити також на N.

console.log(0.17 + 0.24); *// 0.41000000000000003*  
console.log((0.17 \* 100 + 0.24 \* 100) / 100); *// 0.41*

Ще один спосіб - додати, а результат скоротити до певного знаку після коми за допомогою методу toFixed().

console.log(0.17 + 0.24); *// 0.41000000000000003*  
console.log((0.17 + 0.24).toFixed(2)); *// 0.41*

## Клас Math[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/numbers" \l "%D0%BA%D0%BB%D0%B0%D1%81-math" \o "Direct link to heading)

Один із вбудованих класів, який надає набір методів для роботи з числами. Знання всіх методів напам'ять не вимагається, тільки деяких, найбільш корисних.

*// Math.floor(num) - повертає найбільше ціле число,*  
*// менше, або яке дорівнює зазначеному числу*  
console.log(Math.floor(1.7)); *// 1*  
  
*// Math.ceil(num) - повертає найменше ціле число,*  
*// більше, або яке дорівнює зазначеному числу.*  
console.log(Math.ceil(1.2)); *// 2*  
  
*// Math.round(num) - повертає значення числа,*  
*// округленого до найближчого цілого*  
console.log(Math.round(1.2)); *// 1*  
console.log(Math.round(1.5)); *// 2*  
  
*// Math.max(num1, num2, ...) - повертає найбільше ціле число з набору*  
console.log(Math.max(20, 10, 50, 40)); *// 50*  
  
*// Math.min(num1, num2, ...) - повертає найменше ціле число з набору*  
console.log(Math.min(20, 10, 50, 40)); *// 10*  
  
*// Math.pow(base, exponent) - піднесення до степеня*  
console.log(Math.pow(2, 4)); *// 16*  
  
*// Math.random() - повертає псевдовипадкове число в діапазоні [0, 1)*  
console.log(Math.random()); *// випадкове число між 0 і 1*  
console.log(Math.random() \* (10 - 1) + 1); *// псевдовипадкове число від 1 до 10*

# Рядки

**Рядок** - це індексований набір з нуля або більше символів, взятих в одинарні або подвійні лапки.

const username = "Mango";

Важливо пам'ятати, що індексація елементів рядка починається з нуля. Наприклад, в рядку "JavaScript" літера "J" стоїть на позиції з індексом 0, а "t" - під індексом 9.

Вміст рядка не можна змінити, лише прочитати. Тобто, не можна взяти будь-який символ і замінити його, як тільки рядок створений - він такий назавжди. Можна лише створити повністю новий рядок і присвоїти у змінну замість старого.

## Конкатенація рядків[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/strings#%D0%BA%D0%BE%D0%BD%D0%BA%D0%B0%D1%82%D0%B5%D0%BD%D0%B0%D1%86%D1%96%D1%8F-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2)

Якщо застосувати оператор + до рядка і будь-якого іншого типу даних, результатом операції «додавання» буде рядок. Ця операція називається конкатенація, або додавання рядків.

Під час конкатенації, будь-який тип даних приводиться до рядка і зшивається з рядком, але є особливість - послідовність запису операндів.

Послідовність операцій має значення, перетворення типів відбувається тільки в момент операції додавання з рядком, до цього моменту діють звичні правила математики.

const message = "Mango " + "is" + " happy";  
console.log(message); *// Mango is happy*

Подивимось на різний порядок операндів.

console.log(1 + "2"); *// "12"*  
console.log(1 + "2" + 4); *// "124"*  
console.log(1 + 2 + "4"); *// "34"*

В останньому прикладі відбулася математична операція додавання для перших двох чисел 1 і 2, після чого число 3 було перетворено у рядок "3" і зшито з рядком "4".

## Шаблонні рядки[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/strings#%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D0%BD%D1%96-%D1%80%D1%8F%D0%B4%D0%BA%D0%B8)

Шаблонні рядки - це альтернатива конкатенації зі зручнішим синтаксисом. Шаблонний рядок береться у зворотні (косі) лапки, замість подвійних або одинарних, і може містити заповнювачі місця, які позначаються знаком долара і фігурними дужками - ${вираз}.

*// Використовуючи змінні, необхідно скласти рядок з підставленими значеннями*  
const guestName = "Манго";  
const roomNumber = 207;  
const greeting =  
 "Welcome " + guestName + ", your room number is " + roomNumber + "!";  
console.log(greeting); *// "Welcome Mango, your room number is 207!"*

Складати рядки з підставленими значеннями, використовуючи конкатенацію, - дуже незручно. На допомогу приходять шаблонні рядки та інтерполяція.

const guestName = "Манго";  
const roomNumber = 207;  
const greeting = `Welcome ${guestName}, your room number is ${roomNumber}!`;  
console.log(greeting); *// "Welcome Mango, your room number is 207!"*

## Властивості і методи рядків[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/strings#%D0%B2%D0%BB%D0%B0%D1%81%D1%82%D0%B8%D0%B2%D0%BE%D1%81%D1%82%D1%96-%D1%96-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8-%D1%80%D1%8F%D0%B4%D0%BA%D1%96%D0%B2)

У кожного рядка є вбудовані властивості і методи, розглянемо деякі з них.

### Властивість length[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/strings" \l "%D0%B2%D0%BB%D0%B0%D1%81%D1%82%D0%B8%D0%B2%D1%96%D1%81%D1%82%D1%8C-length" \o "Direct link to heading)

Для того, щоб дізнатися довжину рядка, тобто кількість його символів, у всіх рядків є вбудована властивість length, значення якої можна отримати, звернувшись до нього через крапку після імені змінної або рядкового літерала.

const message = "Welcome to Bahamas!";  
console.log(message.length); *// 19*  
console.log("There is nothing impossible to him who will try".length); *// 47*

### Методи toLowerCase() і toUpperCase()[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/strings#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8-tolowercase-%D1%96-touppercase)

Повертають новий рядок у відповідному регістрі, не змінюючи оригінальний рядок.

const message = "Welcome to Bahamas!";  
console.log(message.toLowerCase()); *// "welcome to bahamas!"*  
console.log(message.toUpperCase()); *// "WELCOME TO BAHAMAS!"*  
console.log(message); *// "Welcome to Bahamas!"*

Бувають ситуації, коли всі символи в рядку необхідно перетворити в один регістр, верхній або нижній. Наприклад, для пошуку за ключовим словом, коли користувач вводить рядок 'saMsUng', а порівняти його потрібно з рядком 'samsung' або 'SAMSUNG'.

console.log("saMsUng" === "samsung"); *// false*  
console.log("saMsUng" === "SAMSUNG"); *// false*

Щоб не вимагати абсолютно точне введення, можна зробити «нормалізацію» введеного користувачем рядка, тобто перетворити всі його символи у верхній або нижній регістр. Методи рядка toLowerCase() і toUpperCase() повернуть новий рядок у відповідному регістрі, не змінюючи оригінальний.

const BRAND\_NAME = "SAMSUNG";  
const userInput = "saMsUng";  
const normalizedToUpperCaseInput = userInput.toUpperCase();  
  
console.log(userInput); *// 'saMsUng'*  
console.log(userInput === BRAND\_NAME); *// false*  
console.log(normalizedToUpperCaseInput); *// 'SAMSUNG'*  
console.log(normalizedToUpperCaseInput === BRAND\_NAME); *// true*

### Метод indexOf()[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/strings#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-indexof)

Повертає позицію (індекс), на якій знаходиться перший збіг підрядка або -1, якщо нічого не знайдено.

const message = "Welcome to Bahamas!";  
console.log(message.indexOf("to")); *// 8*  
console.log(message.indexOf("hello")); *// -1*

### Метод includes()[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/strings#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-includes)

Перевіряє, чи міститься підрядок в рядку, повертає буль - true, якщо міститься, і false - в іншому випадку. Регістр символів в рядку і підрядку має значення, оскільки, наприклад, літера "a" не дорівнює літері "А".

const productName = "Ремонтний дроїд";  
  
console.log(productName.includes("н")); *// true*  
console.log(productName.includes("Н")); *// false*  
console.log(productName.includes("дроїд")); *// true*  
console.log(productName.includes("Дроїд")); *// false*  
console.log(productName.includes("Ремонтний")); *// true*  
console.log(productName.includes("ремонтний")); *// false*

ЦІКАВО

Усі методи рядків чутливі до регістру.

### Метод endsWith()[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/strings#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-endswith)

Дозволяє визначити, чи завершується рядок символами (підрядком), зазначеними в дужках, повертаючи true або false.

const jsFileName = "script.js";  
console.log(jsFileName.endsWith(".js")); *// true*  
  
const cssFileName = "styles.css";  
console.log(cssFileName.endsWith(".js")); *// false*

### Методи replace() і replaceAll()[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/strings#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8-replace-%D1%96-replaceall)

Повертає новий рядок, в якому перше (replace) або усі збіги (replaceAll) підрядка замінені на вказане значення.

const jsFileName = "script.js";  
const minifiedJsFileName = jsFileName.replace(".js", ".min.js");  
console.log(minifiedJsFileName); *// "script.min.js"*  
  
const cssFileNames = "styles.css, about.css, portfolio.css";  
const minifiedCssFileNames = cssFileNames.replaceAll(".css", ".min.css");  
console.log(minifiedCssFileNames); *// "styles.min.css, about.min.css, portfolio.min.css"*

### Метод slice()[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/strings#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-slice)

Метод рядків slice (startIndex, endIndex) використовується для створення копії частини або всього рядка. Він робить копію елементів рядка від startIndex і до, але не включно endIndex і повертає новий рядок, не змінюючи оригінал.

const productName = "Repair droid";  
console.log(productName.slice(0, 4)); *// "Repa"*  
console.log(productName.slice(3, 9)); *// "air dr"*  
console.log(productName.slice(0, productName.length)); *// "Repair droid"*  
console.log(productName.slice(7, productName.length)); *// "droid"*

# Логічні оператори

Логічні оператори використовуються для перевірки умов з багатьма виразами, наприклад в операціях порівняння.

## Приведення типів[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/logical-operators#%D0%BF%D1%80%D0%B8%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%BD%D1%8F-%D1%82%D0%B8%D0%BF%D1%96%D0%B2)

У логічних операціях здійснюється приведення типів операндів до true або false. Приведення відбувається, якщо в коді виявлений логічний оператор.

**Truthy** і **Falsy** - терміни, які використовуються для тих значень, які в логічній операції приводяться до true або false, хоча спочатку не були булями.

ЦІКАВО

Запам'ятайте 6 хибних (falsy) значень, що приводяться до false у логічному перетворенні: 0, NaN, null, undefined, порожній рядок і false. Абсолютно все інше приводиться до true.

## Логічні оператори[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/logical-operators#%D0%BB%D0%BE%D0%B3%D1%96%D1%87%D0%BD%D1%96-%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%B8)

Існує три логічні оператори, які використовуються для перевірки виконання множинних виразів.

### Логічне «І»[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/logical-operators#%D0%BB%D0%BE%D0%B3%D1%96%D1%87%D0%BD%D0%B5-%D1%96)

Оператор && приводить всі операнди до буля і повертає значення одного з них. Лівий операнд - якщо його можна привести до false, і правий - в інших випадках.

вираз && вираз

У наступному прикладі обидві умови повернуть true, тому результатом всього виразу буде true - повернеться значення правого операнда.

const age = 20;  
console.log(age > 10 && age < 30); *// true && true -> true*

Якщо хоча б один із операндів буде приведений до false, результатом виразу буде його значення.

const age = 50;  
console.log(age > 10 && age < 30); *// true && false -> false*  
console.log(age > 80 && age < 120); *// false && true -> false*

Тобто, логічне «І» зупиняється на хибному і повертає те, на чому зупинилось, або останній операнд.

console.log(1 && 5); *// true && true -> 5*  
console.log(5 && 1); *// true && true -> 1*  
console.log(0 && 2); *// false && true -> 0*  
console.log(2 && 0); *// true && false -> 0*  
console.log("" && "Mango"); *// false && true -> ""*  
console.log("Mango" && ""); *// true && false -> ""*  
console.log("Mango" && "Poly"); *// true && true -> "Poly"*  
console.log("Poly" && "Mango"); *// true && true -> "Mango"*

ЦІКАВО

Виконуючи логічне «І», правий операнд може не обчислюватися у випадку, якщо лівий був приведений до false.

### Логічне «АБО»[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/logical-operators#%D0%BB%D0%BE%D0%B3%D1%96%D1%87%D0%BD%D0%B5-%D0%B0%D0%B1%D0%BE)

Оператор || приводить всі операнди до буля і повертає значення одного з них. Лівий операнд - якщо його можна привести до true, і правий - в інших випадках.

вираз || вираз

У наступному прикладі умова зліва поверне true, тому результатом всього виразу буде true - повернеться значення першого операнда, яке було приведене до true.

const age = 5;  
console.log(age < 10 || age > 30); *// true || false -> true*

В цьому випадку результатом теж буде true, оскільки хоча б один із операндів, в цьому випадку правий, був приведений до true.

const age = 40;  
console.log(age < 10 || age > 30); *// false || true -> true*

А тут жодна з умов не виконується, тому отримуємо false - значення останнього операнда.

const age = 20;  
console.log(age < 10 || age > 30); *// false || false -> false*

Тобто, логічне «АБО» зупиняється на правді і повертає те, на чому зупинилося або останній операнд.

console.log(true || false); *// true*  
console.log(false || true); *// true*  
console.log(true || true); *// true*  
  
console.log(3 || false); *// 3*  
console.log(false || 3); *// 3*  
console.log(3 || true); *// 3*  
console.log(true || 3); *// true*

ЦІКАВО

Виконуючи логічне «АБО», правий операнд може не обчислюватися у випадку, якщо лівий був приведений до true.

### Логічне «НЕ»[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/logical-operators#%D0%BB%D0%BE%D0%B3%D1%96%D1%87%D0%BD%D0%B5-%D0%BD%D0%B5)

Всі оператори, які ми розглядали до цього, були **бінарними** - містять два операнди: лівий і правий. Логічне «НЕ» - це **унарний** оператор, який виконує операцію над одним операндом з правої сторони.

!вираз

Оператор ! приводить операнд до буля, якщо необхідно, а потім робить інверсію - змінює його на протилежний true -> false або false -> true.

console.log(!true); *// false*  
console.log(!false); *// true*  
console.log(!3); *// !3 -> !true -> false*  
console.log(!"Mango"); *// !"Mango" -> !true -> false*  
console.log(!0); *// !0 -> !false -> true*  
console.log(!""); *// !"" -> !false -> true*  
  
const isOnline = true;  
const isNotOnline = !isOnline; *// !isOnline -> !true -> false*

# Розгалуження

Розгалуження використовуються для виконання різноманітного коду, залежно від умови. Принцип роботи простий - результат умови приводиться до буля true або false, після чого потік програми спрямовується в ту або іншу гілку.

## Інструкція if[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/branching" \l "%D1%96%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%86%D1%96%D1%8F-if" \o "Direct link to heading)

if (умова) {  
 *// тіло if*  
}  
Вхідні дані, які приводяться до буля, називаються **умовою**. Умова ставиться після оператора if в круглих дужках. Якщо умова приводиться до true, то виконується код у фігурних дужках тіла if.

let cost = 0;  
const subscription = "pro";  
  
if (subscription === "pro") {  
 cost = 100;  
}  
  
console.log(cost); *// 100*

Якщо умова приводиться до false, код у фігурних дужках буде пропущений.

let cost = 0;  
const subscription = "free";  
  
if (subscription === "pro") {  
 cost = 100;  
}  
  
console.log(cost); *// 0*

## Інструкція if...else[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/branching" \l "%D1%96%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%86%D1%96%D1%8F-ifelse" \o "Direct link to heading)

if (умова) {  
 *// тіло if*  
} else {  
 *// тіло else*  
}

Розширює синтаксис if таким чином, що якщо умова приводиться до false, виконається код у фігурних дужках після оператора else.

let cost;  
const subscription = "free";  
  
if (subscription === "pro") {  
 cost = 100;  
} else {  
 cost = 0;  
}  
  
console.log(cost); *// 0*

Якщо умова приводиться до true, тіло блока else ігнорується.

let cost;  
const subscription = "pro";  
  
if (subscription === "pro") {  
 cost = 100;  
} else {  
 cost = 0;  
}  
  
console.log(cost); *// 100*

## Інструкція else...if[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/branching" \l "%D1%96%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%86%D1%96%D1%8F-elseif" \o "Direct link to heading)

Конструкція if...else може перевірити і зреагувати на виконання або невиконання лише однієї умови.

Блок else...if дозволяє додати після else ще один оператор if з умовою. В кінці ланцюжка може бути класичний блок else, який виконається лише у тому випадку, якщо жодна умова не приведеться до true.

let cost;  
const subscription = "premium";  
  
if (subscription === "free") {  
 cost = 0;  
} else if (subscription === "pro") {  
 cost = 100;  
} else if (subscription === "premium") {  
 cost = 500;  
} else {  
 console.log("Invalid subscription type");  
}  
  
console.log(cost); *// 500*

При першому true перевірки припиняться і виконається лише один сценарій, який відповідає цьому true. Тому, такий запис варто читати як: шукаю перший збіг умови, ігнорую все інше.

**Тернарний оператор**

Тернарний оператор використовується у якості синтаксично коротшої заміни інструкції if...else, коли одній і тій самій змінній необхідно присвоїти різні значення за умовою.

<умова> ? <вираз\_якщо\_умова\_правдива> : <вираз\_якщо\_умова\_хибна>

Працює за наступною схемою:

* Обчислюється умова.
* Якщо умова правдива, тобто приводиться до true, обчислюється вираз після ?.
* Якщо умова хибна, тобто приводиться до false, обчислюється вираз після :.
* Значення обчисленого виразу повертається у якості результату роботи тернарного оператора.

let type;  
const age = 20;  
  
if (age >= 18) {  
 type = "adult";  
} else {  
 type = "child";  
}  
  
console.log(type); *// "adult"*

Виконаємо рефакторинг, замінивши if...else на тернарний оператор.

const age = 20;  
const type = age >= 18 ? "adult" : "child";  
console.log(type); *// "adult"*

Запишемо операцію пошуку більшого числа.

const num1 = 5;  
const num2 = 10;  
let biggerNumber;  
  
if (num1 > num2) {  
 biggerNumber = num1;  
} else {  
 biggerNumber = num2;  
}  
  
console.log(biggerNumber); *// 10*

Код працює правильно, отримуємо більше число з двох, але це рішення здається занадто громіздким, враховуючи, наскільки проста проблема. Використовуємо тернарний оператор.

const num1 = 5;  
const num2 = 10;  
const biggerNumber = num1 > num2 ? num1 : num2;  
  
console.log(biggerNumber); *// 10*

ЦІКАВО

Тернарний оператор повинен використовуватися у простих операціях присвоєння або повернення. Його використання для опису складних розгалужень - погана практика (антипатерн).

# Інструкція switch

У деяких випадках незручність читання складних розгалужень if...else можна уникнути, використовуючи «плоскіший» синтаксис інструкції розгалуження switch.

Межі застосування switch обмежені задачами з одним загальним запитанням (що порівнювати) і рядом варіантів відповідей (з чим порівнювати).

Його синтаксис складається із блоку switch(значення) - що потрібно порівняти і набору окремих випадків case значення - з чим потрібно порівняти. Для порівняння використовується оператор строгої рівності ===. Тобто, не можна порівняти більше або менше, лише рівність.

switch (значення) {  
 case значення:  
 інструкції;  
 break;  
  
 case значення:  
 інструкції;  
 break;  
  
 default:  
 інструкції;  
}

Значення в блоці switch(значення) - рядок або число, яке порівнюється щодо строгої рівності з усіма значеннями в блоках case значення по черзі, зверху вниз.

Оператор break в кінці кожного блоку case необхідний, щоб перервати подальші перевірки і одразу перейти до коду за switch у тому випадку, коли перевірка рівності повернула true.

Якщо жодного збігу значень не відбулося, необхідно виконати код за замовчуванням, як і в блоці else для інструкції if...else. Для цього, після усіх блоків case додається блок default. Оператор break після блоку default не потрібен, тому що це вже остання операція, яка буде виконана в switch і управління буде передано коду після нього.

let cost;  
const subscription = "premium";  
  
switch (subscription) {  
 case "free":  
 cost = 0;  
 break;  
  
 case "pro":  
 cost = 100;  
 break;  
  
 case "premium":  
 cost = 500;  
 break;  
  
 default:  
 console.log("Invalid subscription type");  
}  
  
console.log(cost); *// 500*

ЦІКАВО

Якщо оператор break відсутній, то після того як виконається будь-яка умова case, усі наступні за ним блоки коду будуть виконуватися один за одним, що може призвести до небажаних наслідків у випадку неправильного застосування.

**Область видимості**

**Область видимості змінних (variable scope)** - доступність змінних в певному місці коду.

Глобальна область видимості використовується за замовчуванням. Кожен має доступ до змінних, оголошених в ній. Наприклад, змінна value оголошена у глобальній області видимості, тобто поза будь-яким блоком, і доступна в будь-якому місці після оголошення.

const value = 5;  
  
if (true) {  
 console.log("Block scope: ", value); *// 5*  
}  
  
console.log("Global scope: ", value); *// 5*

Будь-яка конструкція, яка використовує фігурні дужки {} (умови, цикли, функції тощо), створює нову локальну область видимості, і змінні, оголошені в цій області видимості, з використанням let або const, - недоступні за межами цього блоку.

if (true) {  
 const value = 5;  
 console.log("Block scope: ", value); *// 5*  
}  
  
console.log("Global scope: ", value); *// ReferenceError: value is not defined*

Глибина вкладеності областей видимості - необмежена, і всі вони будуть працювати за одним принципом - область видимості має доступ до всіх змінних, оголошених вище по ієрархії вкладеності, але не може отримати доступ до змінних, оголошених у вкладених областях видимості.

Створимо декілька областей видимості і дамо їм імена для наочності.

* Глобальна область вже присутня, створимо в ній змінну global
* Потім, використовуючи оператор if, створимо блочну область видимості block A
* Всередині області видимості block A поставимо ще один оператор if, який створить вкладену область видимості block B
* На одному рівні з block A, створимо область видимості block C, так само використовуючи оператор if

const global = "global";  
  
if (true) {  
 const blockA = "block A";  
  
 *// Бачимо глобальну + локальну A*  
 console.log(global); *// 'global'*  
 console.log(blockA); *// block A*  
  
 *// Змінні blockB і blockC не знайдені в доступних областях видимості.*  
 *// Буде помилка звернення до змінної.*  
 console.log(blockB); *// ReferenceError: blockB is not defined*  
 console.log(blockC); *// ReferenceError: blockC is not defined*  
  
 if (true) {  
 const blockB = "block B";  
  
 *// Бачимо глобальну + зовнішню A + локальну B*  
 console.log(global); *// global*  
 console.log(blockA); *// block A*  
 console.log(blockB); *// block B*  
  
 *// Змінна blockC не знайдена в доступних областях видимості.*  
 *// Буде помилка звернення до змінної.*  
 console.log(blockC); *// ReferenceError: blockC is not defined*  
 }  
}  
  
if (true) {  
 const blockC = "block C";  
  
 *// Бачимо глобальну + локальну C*  
 console.log(global); *// global*  
 console.log(blockC); *// block C*  
  
 *// Змінні blockA і blockB не знайдені в доступних областях видимості.*  
 *// Буде помилка звернення до змінної.*  
 console.log(blockA); *// ReferenceError: blockA is not defined*  
 console.log(blockB); *// ReferenceError: blockB is not defined*  
}  
  
*// Бачимо лише глобальну*  
console.log(global); *// global*  
  
*// Змінні blockA, blockB і blockC не знайдені в доступних областях видимості.*  
*// Буде помилка звернення до змінної.*  
console.log(blockA); *// ReferenceError: blockA is not defined*  
console.log(blockB); *// ReferenceError: blockB is not defined*  
console.log(blockC); *// ReferenceError: blockC is not defined*

ЦІКАВО

Будьте уважні у разі використання блочних областей видимості і змінних, оголошених в них. Саме ця помилка, разом із неуважністю, часто стають головним болем для початківця.

# Цикли

Часте завдання програмування - багаторазове виконання однотипної дії. Наприклад, вивести клієнтів зі списку один за одним, або перебрати суми зарплат, і для кожної виконати однаковий код. Саме для таких цілей - багаторазового повторення однієї частини коду, використовуються цикли.

* **Цикл** - керуюча конструкція у високорівневих мовах програмування, призначена для організації багаторазового виконання набору інструкцій.
* **Тіло циклу** - послідовність інструкцій, призначена для багаторазового виконання.
* **Ітерація** - одноразове виконання тіла циклу.
* **Умова виходу** - вираз, що визначає, чи буде в черговий раз виконуватися ітерація, або цикл завершиться.
* **Лічильник** - змінна, що зберігає поточний номер ітерації. Цикл не обов'язково містить лічильник, і він не повинен бути один, умова виходу з циклу може залежати від декількох змінних, що змінюються в циклі.

## Цикл while[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/loops" \l "%D1%86%D0%B8%D0%BA%D0%BB-while" \o "Direct link to heading)

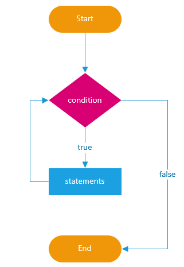
**Цикл з передумовою** — цикл, який виконується, доки правдива якась умова, зазначена до його початку. Ця умова перевіряється до виконання тіла циклу, тому тіло може не виконатися жодного разу, якщо умова від самого початку - хибна.

while (condition) {  
 *// код, тіло циклу (statement)*  
}

Конструкція while створює цикл, який виконує блок коду, доки умова перевірки оцінюється як true.

* condition, тобто умова, оцінюється перед кожною ітерацією циклу.
* Якщо condition оцінюється як true, оператор while виконує statement.
* Якщо condition оцінюється як false, виконання циклу переривається і скрипт продовжує виконувати інструкції після циклу while.

Блок-схема ілюструє цикл while.



Створимо лічильник.

let counter = 0;  
  
while (counter < 10) {  
 console.log("counter: ", counter);  
 counter += 1;  
}

Будемо заповнювати місця в готелі доти, доки поточна кількість клієнтів не буде дорівнювати максимально можливій.

let clientCounter = 18;  
const maxClients = 25;  
  
while (clientCounter < maxClients) {  
 console.log(clientCounter);  
 clientCounter += 1;  
}

## Цикл do...while[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/loops" \l "%D1%86%D0%B8%D0%BA%D0%BB-dowhile" \o "Direct link to heading)

**Цикл з постумовою** - цикл, в якому умова перевіряється після виконання тіла циклу. З цього випливає, що тіло завжди виконується хоча б один раз.

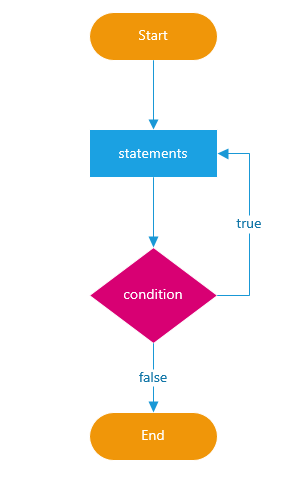
do {  
 *// statement*  
} while (condition);

Конструкція do...while створює цикл, який виконує блок коду доти, доки condition не поверне false.

На відміну від циклу while, цикл do...while завжди виконує statement мінімум один раз, перш ніж він оцінить condition.

Всередині циклу потрібно внести зміни в деяку змінну, щоб переконатися, що вираз дорівнює false після ітерацій. Інакше буде нескінченний цикл.

Блок-схема ілюструє цикл do-while



let password = "";  
  
do {  
 password = prompt("Введіть пароль довший 4-х символів", "");  
} while (password.length < 5);  
  
console.log("Ввели пароль: ", password);

## Цикл for[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/loops" \l "%D1%86%D0%B8%D0%BA%D0%BB-for" \o "Direct link to heading)

**Цикл з лічильником** - цикл, в якому певна змінна змінює своє значення із заданого початкового до кінцевого значення з деяким кроком, і для кожного значення цієї змінної тіло циклу виконується один раз.

В більшості процедурних мов програмування реалізується конструкцією for, в якій зазначається лічильник, необхідна кількість ітерацій і крок, з яким змінюється лічильник.

for (initialization; condition; post-expression) {  
 // statements  
}

Алгоритм виконання циклу for:

* **Ініціалізація (initialization)** - виконується один раз до початку циклу. Використовується для створення змінної-лічильника і зазначення її початкового значення.
* **Умова (condition)** - вираз, що оцінюється перед кожною ітерацією (повторенням) циклу. Тіло циклу виконується тільки тоді, коли вираз приводиться до true. Цикл завершується, якщо значення буде false.
* **Тіло (statements)** - набір інструкцій для виконання на кожному повторенні. Виконується, якщо вираз умови приводиться до true.
* **Пост-вираз (post-expression)** - виконується в кінці кожного повторення циклу, до перевірки умови. Використовується для оновлення змінної-лічильника.

Змінні-лічильники, за традицією, називаються літерами i, j і k.

for (let i = 0; i <= 20; i += 5) {  
 console.log(i);  
}

У прикладі оголошується змінна-лічильник i, ініціалізується значенням 0 і цикл виконується доти, доки i <= 20, тобто умова приводиться до true. Після кожної ітерації лічильник збільшується на 5.

Порахуємо суму чисел до певного значення.

const target = 3;  
let sum = 0;  
  
for (let i = 0; i <= target; i += 1) {  
 sum += i;  
}  
  
console.log(sum);

Згадаємо про операцію a % b і виведемо остачу від ділення за допомогою циклу.

const max = 10;  
for (let i = 0; i < max; i += 1) {  
 console.log(`${max} % ${i} = `, max % i);  
}

## Оператор break[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/loops" \l "%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80-break" \o "Direct link to heading)

Перервати виконання циклу можна в будь-який момент. Для цього існує оператор break, який повністю припиняє виконання циклу і передає управління на рядок після його тіла.

Знайдемо число 3. Щойно виконається умова if, цикл припинить своє виконання (буде перерваний).

for (let i = 0; i <= 5; i += 1) {  
 console.log(i);  
  
 if (i === 3) {  
 console.log("Знайшли число 3, перериваємо виконання циклу");  
 break;  
 }  
}  
  
console.log("Лог після циклу");

## Оператор continue[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-01/loops" \l "%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80-continue" \o "Direct link to heading)

Перериває не увесь цикл, а тільки виконання поточної ітерації. Його використовують, якщо зрозуміло, що на поточній ітерації циклу більше немає що робити або взагалі не потрібно нічого робити, і час переходити до наступної ітерації.

Використовуємо цикл для введення тільки непарних чисел. Для парних i спрацьовує continue, виконання тіла припиняється і управління передається до наступної ітерації.

const number = 10;  
  
for (let i = 0; i < number; i += 1) {  
 if (i % 2 === 0) {  
 continue;  
 }  
  
 console.log("Непарне i: ", i); *// 1, 3, 5, 7, 9*  
}

# Масиви

**Масив** - структура даних для зберігання і маніпулювання колекцією індексованих значень. Використовується для зберігання впорядкованих колекцій даних, наприклад, списку курортів, товарів, клієнтів в готелі тощо.

## Створення[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/arrays/#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F)

Масив оголошується і береться у квадратні дужки [] - літералом масиву. Всередині дужок кожен елемент масиву розділяється комою.

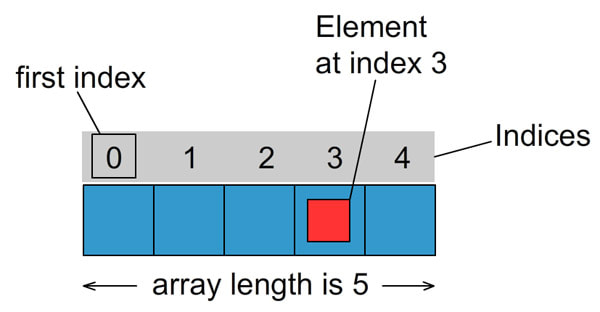
const clients = ["Mango", "Poly", "Ajax"];

## Доступ до елементів[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/arrays/#%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF-%D0%B4%D0%BE-%D0%B5%D0%BB%D0%B5%D0%BC%D0%B5%D0%BD%D1%82%D1%96%D0%B2)

Для доступу до значення елемента масиву використовується синтаксис квадратних дужок масив[індекс]. Між іменем змінної, що зберігає масив, і квадратними дужками не повинно бути пробілу.

const clients = ["Mango", "Poly", "Ajax"];  
  
*// Зазначаючи в дужках індекс елемента, ми отримуємо його значення*  
console.log(clients[0]); *// Mango*  
console.log(clients[1]); *// Poly*  
console.log(clients[2]); *// Ajax*

Індексація елементів масиву починається з **нуля**.



## Перевизначення[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/arrays/" \l "%D0%BF%D0%B5%D1%80%D0%B5%D0%B2%D0%B8%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%BD%D1%8F" \o "Direct link to heading)

На відміну від рядків, елементи масиву можна змінювати, звернувшись до них за індексом і присвоївши інше значення.

const clients = ["Mango", "Poly", "Ajax"];  
clients[0] = "Kiwi";  
clients[1] = "Pango";  
console.log(clients); *// ["Kiwi", "Pango", "Ajax"]*

## Довжина масиву[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/arrays/#%D0%B4%D0%BE%D0%B2%D0%B6%D0%B8%D0%BD%D0%B0-%D0%BC%D0%B0%D1%81%D0%B8%D0%B2%D1%83)

Довжина масиву, тобто кількість його елементів, зберігається у властивості length. Це динамічна величина, яка змінюється автоматично під час додавання або видалення елементів.

const clients = ["Mango", "Poly", "Ajax"];  
console.log(clients.length); *// 3*

## Індекс останнього елемента[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/arrays/#%D1%96%D0%BD%D0%B4%D0%B5%D0%BA%D1%81-%D0%BE%D1%81%D1%82%D0%B0%D0%BD%D0%BD%D1%8C%D0%BE%D0%B3%D0%BE-%D0%B5%D0%BB%D0%B5%D0%BC%D0%B5%D0%BD%D1%82%D0%B0)

Найчастіше, ми заздалегідь не знаємо яка буде довжина масиву в коді. Для того, щоб отримати значення останнього елемента, застосовується наступний підхід - довжина масиву завжди на одиницю більша, ніж індекс останнього елемента. Використовуючи формулу довжина\_масиву - 1, можна отримати значення останнього елемента масиву довільної довжини.

const clients = ["Mango", "Poly", "Ajax"];  
const lastElementIndex = clients.length - 1;  
console.log(lastElementIndex); *// 2*  
console.log(clients[lastElementIndex]); *// "Ajax"*

# Ітерація по масиву

Цикл for можна використовувати для ітерації по масиву, тобто «перебрати» його поелементно.

const clients = ["Mango", "Ajax", "Poly"];  
  
for (let i = 0; i < clients.length; i += 1) {  
 console.log(clients[i]);  
}

Для доступу до елементів використовується синтаксис квадратних дужок масив[індекс], де індекс - це значення лічильника циклу від 0 і до останнього індексу масиву, тобто менше, але не дорівнює його довжині.

## Цикл for...of[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/iteration" \l "%D1%86%D0%B8%D0%BA%D0%BB-forof" \o "Direct link to heading)

Конструкція for...of оголошує цикл, що перебирає ітерабельні об'єкти, як-от масиви і рядки. Тіло циклу буде виконуватися для значення кожного елемента. Це хороша заміна циклу for, якщо не потрібен доступ до лічильника ітерації.

for (const variable of iterable) {  
 *// тіло циклу*  
}

* variable — змінна, яка буде зберігати значення елемента на кожній ітерації.
* iterable — колекція, яка містить ітерабельні (що можна порахувати) елементи, наприклад масив.

const clients = ["Mango", "Ajax", "Poly"];  
  
for (const client of clients) {  
 console.log(client);  
}  
  
const string = "javascript";  
  
for (const character of string) {  
 console.log(character);  
}

## Оператори break і continue[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/iteration" \l "%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%D0%B8-break-%D1%96-continue" \o "Direct link to heading)

Будемо шукати ім'я клієнта в масиві імен, якщо знайшли - перериваємо цикл, оскільки немає сенсу шукати далі, імена у нас унікальні.

const clients = ["Mango", "Poly", "Ajax"];  
const clientNameToFind = "Poly";  
let message;  
  
for (const client of clients) {  
 *// На кожній ітерації будемо перевіряти чи збігається елемент масиву з*  
 *// іменем клієнта. Якщо збігається - записуємо в message повідомлення*  
 *// про успіх і робимо break, щоб далі не шукати*  
 if (client === clientNameToFind) {  
 message = "Клієнт з таким ім'ям є в базі даних!";  
 break;  
 }  
  
 *// Якщо вони не збігаються - записуємо в message повідомлення про відсутність імені*  
 message = "Клієнт з таким ім'ям відсутній в базі даних!";  
}  
  
console.log(message); *// "Клієнт з таким ім'ям є в базі даних!"*

Можна на початку задати message значення невдачі пошуку, а в циклі перезаписати його на успіх, якщо знайшли ім'я. Але break все одно нам знадобиться, оскільки, якщо у нас масив із 10000 клієнтів, а потрібний нам знаходиться на 2 позиції, то немає абсолютно жодного сенсу перебирати інші 9998 елементи.

const clients = ["Mango", "Poly", "Ajax"];  
const clientNameToFind = "Poly";  
let message = "Клієнт з таким ім'ям відсутній в базі даних!";  
  
for (const client of clients) {  
 if (client === clientNameToFind) {  
 message = "Клієнт з таким ім'ям є в базі даних!";  
 break;  
 }  
 *// Якщо не збігається, то на цій ітерації нічого не робимо*  
}  
  
console.log(message); *// Клієнт з таким ім'ям є в базі даних!*

Використовуємо цикл для виведення тільки чисел, більших за певне значення.

const numbers = [1, 3, 14, 18, 4, 7, 29, 6, 34];  
const threshold = 15;  
  
*// Для чисел, менших ніж порогове значення, спрацьовує continue, виконання тіла*  
*// припиняється і управління передається на наступну ітерацію.*  
for (let i = 0; i < numbers.length; i += 1) {  
 if (numbers[i] < threshold) {  
 continue;  
 }  
  
 console.log(`Число більше за ${threshold}: ${numbers[i]}`); *// 18, 29, 34*  
}

# Методи масиву

## Методи split() і join()[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/methods#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8-split-%D1%96-join)

Метод split(delimeter) перетворює рядок в масив, «розбиваючи» його роздільником delimeter. Якщо роздільник - це порожній рядок, то створиться масив окремих символів. Роздільником може бути один або декілька символів.

const name = "Mango";  
console.log(name.split("")); *// ["M", "a", "n", "g", "o"]*  
  
const message = "JavaScript - це цікаво";  
console.log(message.split(" ")); *// ["JavaScript", "-", "це", "цікаво"]*

Метод масивів join(delimeter) об'єднує елементи масиву у рядок. У рядку елементи будуть розділені символом або групою символів, зазначених в delimeter. Тобто ця операція протилежна методу рядків split(delimeter).

const words = ["JavaScript", "це", "цікаво"];  
console.log(words.join("")); *// "JavaScriptцецікаво"*  
console.log(words.join(" ")); *// "JavaScript це цікаво"*  
console.log(words.join("-")); *// "JavaScript-це-цікаво"*

## Метод indexOf()[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/methods#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-indexof)

indexOf(value) повертає перший індекс, в якому елемент зі значенням value був знайдений в масиві, або число -1, якщо такий елемент відсутній. Використовуйте indexOf тоді, коли необхідно отримати сам індекс елемента.

const clients = ["Mango", "Ajax", "Poly", "Kiwi"];  
console.log(clients.indexOf("Poly")); *// 2*  
console.log(clients.indexOf("Monkong")); *// -1*

## Метод includes()[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/methods#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-includes)

includes(value) перевіряє, чи містить масив елемент зі значенням value і повертає true або false відповідно. Застосування цього методу корисне в ситуаціях, коли необхідно перевірити, чи є елемент в масиві і не важлива його позиція (індекс).

const clients = ["Mango", "Ajax", "Poly", "Kiwi"];  
console.log(clients.includes("Poly")); *// true*  
console.log(clients.includes("Monkong")); *// false*

### Перевірка багатьох умов з includes()[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/methods#%D0%BF%D0%B5%D1%80%D0%B5%D0%B2%D1%96%D1%80%D0%BA%D0%B0-%D0%B1%D0%B0%D0%B3%D0%B0%D1%82%D1%8C%D0%BE%D1%85-%D1%83%D0%BC%D0%BE%D0%B2-%D0%B7-includes)

На перший погляд код наступного прикладу виглядає добре.

const fruit = "apple";  
  
if (fruit === "apple" || fruit === "strawberry") {  
 console.log("It is a red fruit!");  
}

Однак, що робити, якщо у нас буде більше червоних фруктів, наприклад, ще вишня (cherry) або журавлина (cranberries)? Чи будемо ми розширювати умову за допомогою додаткових ||?

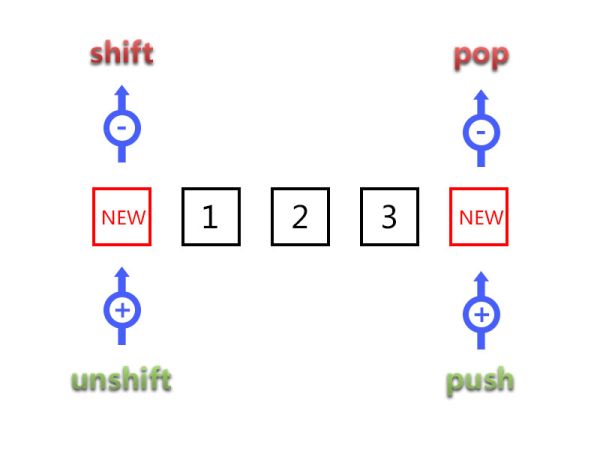
const fruit = "apple";  
  
if (  
 fruit === "apple" ||  
 fruit === "strawberry" ||  
 fruit === "cherry" ||  
 fruit === "cranberries"  
) {  
 console.log("It is a red fruit!");  
}

Можемо переписати умову, використовуючи includes(), це дуже просто і масштабовано.

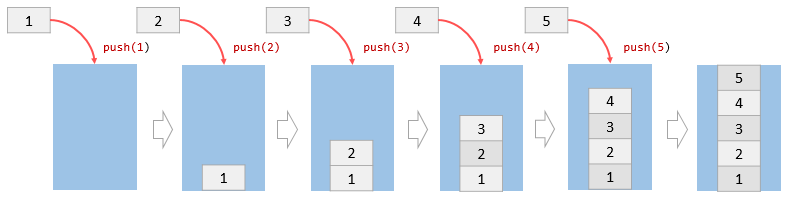
*// Виносимо варіанти в масив*  
const redFruits = ["apple", "strawberry", "cherry", "cranberries"];  
const fruit = "cherry";  
*// Перевіряємо присутність елемента*  
const hasFruit = redFruits.includes(fruit);  
  
if (hasFruit) {  
 console.log(`${fruit} is a red fruit!`);  
}

## Методи push() і pop()[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/methods#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B8-push-%D1%96-pop)

Додають або видаляють крайні елементи масиву. Працюють тільки з крайнім лівим і крайнім правим елементом, і не можуть вставити або видалити елемент з довільної позиції.

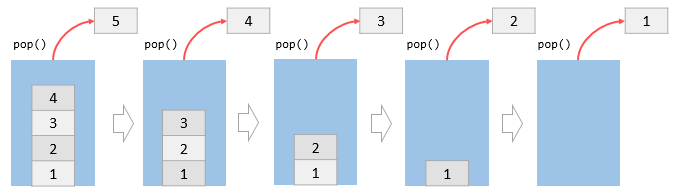


Метод push() додає один або декілька елементів наприкінці масиву, без необхідності зазначати індекси елементів, що додаються. Повертає довжину масиву після додавання елементів.



const numbers = [];  
  
numbers.push(1);  
console.log(numbers); *// [1]*  
  
numbers.push(2);  
console.log(numbers); *// [1, 2]*  
  
numbers.push(3);  
console.log(numbers); *// [1, 2, 3]*  
  
numbers.push(4);  
console.log(numbers); *// [1, 2, 3, 4]*  
  
numbers.push(5);  
console.log(numbers); *// [1, 2, 3, 4, 5]*

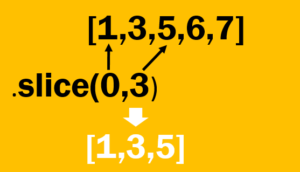
Метод pop() видаляє останній елемент з кінця масиву і повертає видалений елемент. Якщо масив порожній, метод повертає undefined.



const numbers = [1, 2, 3, 4, 5];  
  
console.log(numbers.pop()); *// 5*  
console.log(numbers); *// [1, 2, 3, 4]*  
  
console.log(numbers.pop()); *// 4*  
console.log(numbers); *// [1, 2, 3]*  
  
console.log(numbers.pop()); *// 3*  
console.log(numbers); *// [1, 2]*  
  
console.log(numbers.pop()); *// 2*  
console.log(numbers); *// [1]*  
  
console.log(numbers.pop()); *// 1*  
console.log(numbers); *// []*

## Метод slice()[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/methods#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-slice)

slice(begin, end) повертає новий масив, що містить копію частини вихідного масиву, не змінюючи його. Копія створюється з begin і до, але не включно, end - індекси елементів вихідного масиву.



const clients = ["Mango", "Ajax", "Poly", "Kiwi"];  
console.log(clients.slice(1, 3)); *// ["Ajax", "Poly"]*

Якщо begin і end не зазначені, буде створена повна копія вихідного масиву.

const clients = ["Mango", "Ajax", "Poly", "Kiwi"];  
console.log(clients.slice()); *// ["Mango", Ajax", "Poly", "Kiwi"]*

Якщо не зазначено end, копіювання буде зі start і до кінця вихідного масиву.

const clients = ["Mango", "Ajax", "Poly", "Kiwi"];  
console.log(clients.slice(1)); *// ["Ajax", "Poly", "Kiwi"]*  
console.log(clients.slice(2)); *// ["Poly", "Kiwi"]*

Якщо значення start від'ємне, а end не зазначено - будуть скопійовані останні start елементи

const clients = ["Mango", "Ajax", "Poly", "Kiwi"];  
console.log(clients.slice(-2)); *// ["Poly", "Kiwi"]*

## Метод splice()[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/methods#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-splice)

Швейцарський ніж для роботи з масивами, якщо вихідний масив необхідно змінити. Видаляє, додає і замінює елементи у довільному місці масиву.

### Видалення[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/methods#%D0%B2%D0%B8%D0%B4%D0%B0%D0%BB%D0%B5%D0%BD%D0%BD%D1%8F)

Щоб видалити елементи в масиві, передаються два аргументи.

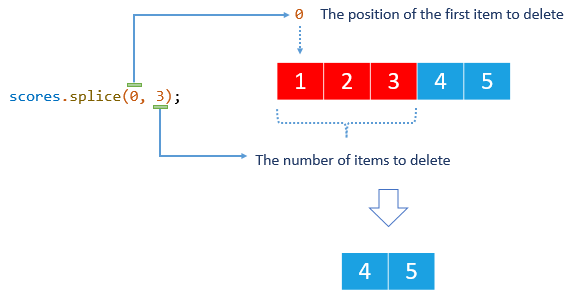
splice(position, num)

* position - вказує на позицію (індекс) першого елемента для видалення
* num - визначає кількість елементів, що видаляються

Метод splice змінює вихідний масив і повертає масив, що містить видалені елементи. Наприклад, у нас є масив оцінок, який містить п'ять чисел від 1 до 5.

const scores = [1, 2, 3, 4, 5];  
  
*// Видаляємо три елементи масиву, починаючи з першого елемента (індекс 0)*  
const deletedScores = scores.splice(0, 3);  
  
*// Тепер масив scores містить два елементи*  
console.log(scores); *// [4, 5]*  
  
*// А масив deletedScores містить три видалені елементи*  
console.log(deletedScores); *// [1, 2, 3]*

На зображенні показаний виклик методу score.splice (0, 3) з прикладу.



ЦІКАВО

На практиці, значення, що повертається (масив видалених елементів), використовується рідко. Переважно, просто необхідно видалити елементи з масиву.

### Додавання[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/methods#%D0%B4%D0%BE%D0%B4%D0%B0%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)

Для того, щоб додати один або декілька елементів в масив, необхідно передати три або більше аргументи, за такої умови, другий аргумент повинен дорівнювати нулю.

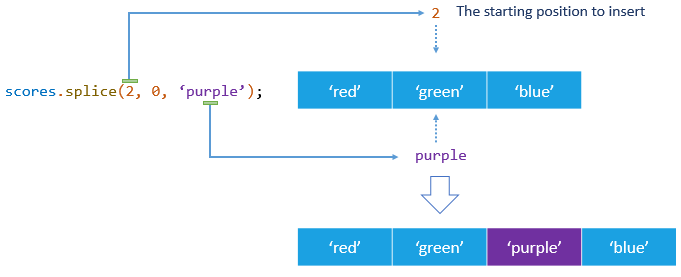
splice(position, 0, new\_element\_1, new\_element\_2, ...)

* Аргумент position вказує початкову позицію в масиві, куди будуть вставлені нові елементи.
* Другий аргумент - це нуль, він говорить методу не видаляти елементи в місці додавання нових.
* Третій, четвертий і всі наступні аргументи - це нові елементи, які додаються в масив.

Наприклад, у нас є масив з назвами квітів у вигляді рядків. Додамо новий колір перед елементом з індексом 2.

const colors = ["red", "green", "blue"];  
  
colors.splice(2, 0, "purple");  
console.log(colors); *// ["red", "green", "purple", "blue"]*

На малюнку показаний виклик методу colors.splice(2, 0, 'purple') з прикладу.



Можна додати довільну кількість елементів, передавши четвертий, п'ятий аргумент тощо.

const colors = ["red", "green", "blue"];  
  
colors.splice(1, 0, "yellow", "pink");  
console.log(colors); *// ["red", "yellow", "pink", "green", "blue"]*

### Заміна[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/methods#%D0%B7%D0%B0%D0%BC%D1%96%D0%BD%D0%B0)

Заміна - це операція додавання, в якій видаляються елементи в місці додавання нових. Для цього необхідно передати мінімум три аргументи. Кількість елементів, що видаляються і додаються, може не збігатися.

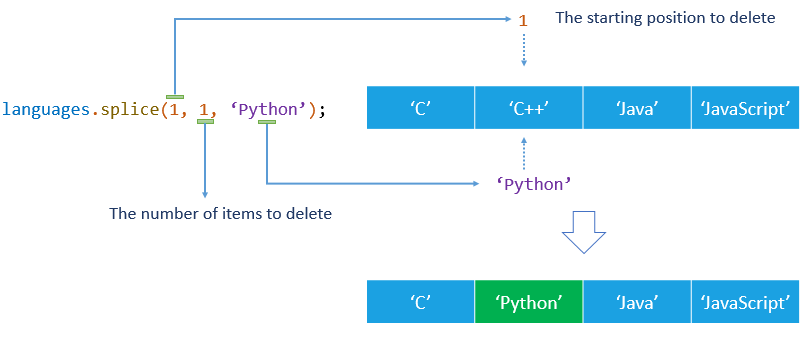
splice(position, num, new\_element\_1, new\_element\_2, ...)

* position - вказує на позицію (індекс) першого елемента для видалення
* num - визначає кількість елементів, що видаляються
* Третій, четвертий і всі наступні аргументи - це нові елементи, які додаються в масив.

Наприклад, у нас є масив мов програмування з чотирьох елементів.

const languages = ["C", "C++", "Java", "JavaScript"];  
  
*// Заміняємо елемент з індексом 1 на новий*  
languages.splice(1, 1, "Python");  
console.log(languages); *// ["C", "Python", "Java", "JavaScript"]*  
  
*// Заміняємо один елемент (з індексом 2) на декілька*  
languages.splice(2, 1, "C#", "Swift", "Go");  
console.log(languages); *// ["C", "Python", "C#", "Swift", "Go", "JavaScript"]*

На зображенні показаний виклик методу languages.splice(1, 1, 'Python') з прикладу.



## Метод concat()[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/methods#%D0%BC%D0%B5%D1%82%D0%BE%D0%B4-concat)

Об'єднує два або більше масивів в один. Він не змінює масив, на якому викликається, а повертає новий. Порядок аргументів методу впливає на порядок елементів нового масиву.

const oldClients = ["Mango", "Ajax", "Poly", "Kiwi"];  
const newClients = ["Monkong", "Singu"];  
  
const allClientsWithOldFirst = oldClients.concat(newClients);  
console.log(allClientsWithOldFirst); *// ["Mango", "Ajax", "Poly", "Kiwi", "Monkong", "Singu"]*  
  
const allClientsWithNewFirst = newClients.concat(oldClients);  
console.log(allClientsWithNewFirst); *// ["Monkong", "Singu", "Mango", "Ajax", "Poly", "Kiwi"]*  
  
console.log(oldClients); *// ["Mango", "Ajax", "Poly", "Kiwi"]*  
console.log(newClients); *// ["Monkong", "Singu"]*

# Присвоєння за посиланням і за значенням

Фундаментальною відмінністю складних типів від примітивів є те, як вони зберігаються і копіюються. Примітиви: рядки, числа, булі, null і undefined, під час присвоєння повністю копіюються **за значенням (by value)**.

Зі складними типами - все по-іншому. У змінній, якій присвоєно масив або об'єкт, зберігається не саме значення, а адреса його місця в пам'яті, іншими словами - посилання (вказівник) на нього і вони передаються **за посиланням (by reference)**.

Уявімо змінну у вигляді аркушу паперу. Її значення ми уявимо як запис на цьому аркуші.

Якщо ми захочемо повідомити зміст цього запису користувачам, то можемо це зробити наступним чином - зробити фізичні копії і вручити кожному, тобто зробити **багато** незалежних копій (присвоєння за значенням).

Або покласти аркуш в зачиненій кімнаті і дати користувачам ключ від цієї кімнати, тобто **один** екземпляр із загальним доступом (присвоєння за посиланням).

Тепер змінимо дані на аркуші паперу - значення змінної. Очевидно, що відвідувачі кімнати завжди будуть бачити зміни, які ми вносимо, оскільки змінюється оригінал і вони мають до нього доступ. І також очевидно, що власники паперових копій не помітять змін, дивлячись на свої копії.

За умови передачі за значенням, змінним виділяється нова комірка пам'яті і в неї копіюються дані. Аналогія з багатьма копіями паперового аркушу має цілком реальне втілення, окремий аркуш для кожної копії.

За умови передачі за посиланням, замість створення нового об'єкта, змінній присвоюється посилання (вказівник) на вже існуючий об'єкт, тобто на його місце в пам'яті. Таким чином, декілька змінних можуть вказувати на один і той самий об'єкт, за аналогією із закритою кімнатою, вони мають ключ доступу до оригіналу аркушу.

Усі примітивні типи присвоюються за значенням, тобто створюється копія.

let a = 5;  
*// Присвоєння за значенням, в пам'яті буде створена ще*  
*// одна комірка, в яку буде скопійоване значення 5*  
let b = a;  
console.log(a); *// 5*  
console.log(b); *// 5*  
  
*// Змінимо значення a*  
a = 10;  
console.log(a); *// 10*  
*// Значення b не змінилося, оскільки це окрема копія*  
console.log(b); *// 5*

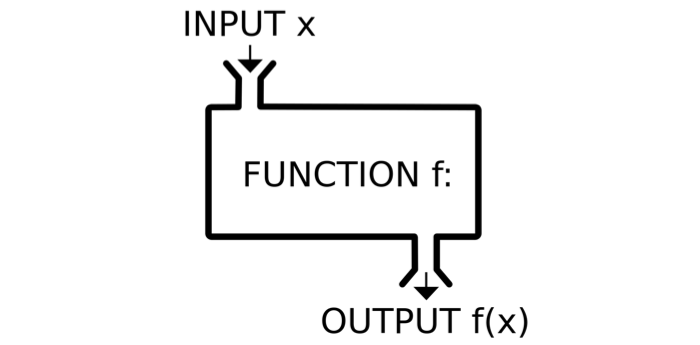
Складні типи - об'єкти, масиви, функції присвоюються за посиланням, тобто змінна просто отримує посилання на вже існуючий об'єкт.

const a = ["Mango"];  
*// Оскільки a - це масив, в b записується посилання на вже існуючий*  
*// масив в пам'яті. Тепер a і b вказують на той самий масив.*  
const b = a;  
console.log(a); *// ["Mango"]*  
console.log(b); *// ["Mango"]*  
  
*// Змінимо масив, додавши ще один елемент, використовуючи вказівник з a*  
a.push("Poly");  
console.log(a); *// ["Mango", "Poly"]*  
  
*// b також змінилось, тому що b, як і a,*  
*// просто містить посилання на те ж саме місце в пам'яті*  
console.log(b); *// ["Mango", "Poly"]*  
  
*// Результат повторюється*  
b.push("Ajax");  
console.log(a); *// ["Mango", "Poly", "Ajax"]*  
console.log(b); *// ["Mango", "Poly", "Ajax"]*

# Функції

**Функція** - це підпрограма, незалежна частина коду, призначена для багаторазового виконання конкретної задачі з різними початковими значеннями. Функції дозволяють структурувати великі програми, зменшують повторення та ізолюють код.

Функцію можна уявити у вигляді чорного ящика: вона отримує щось на вході (дані), і повертає щось на виході (результат виконання коду всередині неї).



## Оголошення функції[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/functions#%D0%BE%D0%B3%D0%BE%D0%BB%D0%BE%D1%88%D0%B5%D0%BD%D0%BD%D1%8F-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%97)

*// 1. Оголошення функції multiply*  
function multiply() {  
 *// Тіло функції*  
 console.log("Це лог на момент виклику функції multiply");  
}  
  
*// 2. Виклики функції multiply*  
multiply(); *// 'Це лог на момент виклику функції multiply'*  
multiply(); *// 'Це лог на момент виклику функції multiply'*  
multiply(); *// 'Це лог на момент виклику функції multiply'*

Оголошення функції (**function declaration**) починається з ключового слова function, після якого стоїть ім'я - дієслово, що відповідає на запитання **«Що зробити?»** і пара круглих дужок.

Тіло функції береться у фігурні дужки {} і містить інструкції, які необхідно виконати на момент її **виклику**. Потім, коли необхідно, функція **викликається** за допомогою імені і пари круглих дужок.

## Параметри та аргументи[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/functions#%D0%BF%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D0%B8-%D1%82%D0%B0-%D0%B0%D1%80%D0%B3%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B8)

В круглих дужках після імені функції зазначаються **параметри** - перелік даних, які функція очікує на момент виклику.

*// Оголошення параметрів x, y, z*  
function multiply(x, y, z) {  
 console.log(`Результат множення дорівнює ${x \* y \* z}`);  
}

Параметри - це локальні змінні, доступні тільки у тілі функції. Вони розділяються комами. Параметрів може бути декілька, або взагалі не бути, у такому випадку записуються просто порожні круглі дужки.

ЦІКАВО

Параметри будуть створюватися кожного разу під час виконання функції, і їх окремі інкарнації жодним чином один з одним не пов'язані.

На момент виклику функції, в круглих дужках можна передати **аргументи** - значення для оголошених параметрів функції.

*// 1. Оголошення параметрів x, y, z*  
function multiply(x, y, z) {  
 console.log(`Результат множення дорівнює ${x \* y \* z}`);  
}  
  
*// 2. Передача аргументів*  
multiply(2, 3, 5); *// Результат множення дорівнює 30*  
multiply(4, 8, 12); *// Результат множення дорівнює 384*  
multiply(17, 6, 25); *// Результат множення дорівнює 2550*

ЦІКАВО

Порядок передачі аргументів повинен відповідати порядку оголошених параметрів: значення першого аргументу буде присвоєно першому параметру, другого аргументу - другому параметру тощо. Якщо параметрів буде більше, ніж аргументів, то параметрам без значень буде присвоєно undefined.

## Повернення значення[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/functions#%D0%BF%D0%BE%D0%B2%D0%B5%D1%80%D0%BD%D0%B5%D0%BD%D0%BD%D1%8F-%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%BD%D1%8F)

Оператор return використовується для передачі значення з тіла функції у зовнішній код. Коли інтерпретатор зустрічає return, він відразу ж виходить з функції (припиняє її виконання), і повертає вказане значення у те місце коду, де була викликана функція.

function multiply(x, y, z) {  
 console.log("Код до return виконується звичайним чином");  
  
 *// Повертаємо результат виразу множення*  
 return x \* y \* z;  
  
 console.log("Цей лог ніколи не виконається, він стоїть після return");  
}  
  
*// Результат роботи функції можна зберегти у змінну*  
let result = multiply(2, 3, 5);  
console.log(result); *// 30*  
  
result = multiply(4, 8, 12);  
console.log(result); *// 384*  
  
result = multiply(17, 6, 25);  
console.log(result); *// 2550*

ЦІКАВО

Оператор return без явно вказаного значення повертає спеціальне значення undefined. За відсутності return в тілі функції, вона все одно поверне undefined.

## Порядок виконання коду[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/functions#%D0%BF%D0%BE%D1%80%D1%8F%D0%B4%D0%BE%D0%BA-%D0%B2%D0%B8%D0%BA%D0%BE%D0%BD%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BA%D0%BE%D0%B4%D1%83)

Коли інтерпретатор зустрічає виклик функції (або методу), він призупиняє виконання поточного коду і починає виконувати код з тіла функції. Після того як увесь код функції буде виконаний, інтерпретатор виходить з тіла функції, повертаючись у те місце, звідки прийшов, і продовжує виконувати код, наступний після виклику функції.

function multiply(x, y, z) {  
 console.log(`Результат множення дорівнює ${x \* y \* z}`);  
}  
  
console.log("Лог до виклику функції multiply");  
multiply(2, 3, 5); *// Результат множення дорівнює 30*  
console.log("Лог після виклику функції multiply");  
  
*// Послідовність логів в консолі*  
*// "Лог до виклику функції multiply"*  
*// "Результат множення дорівнює 30"*  
*// "Лог після виклику функції multiply"*

## Параметри за замовчуванням[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/functions#%D0%BF%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D0%B8-%D0%B7%D0%B0-%D0%B7%D0%B0%D0%BC%D0%BE%D0%B2%D1%87%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F%D0%BC)

Іноді необхідно оголосити функцію, у параметрів якої будуть значення, відмінні від undefined, навіть якщо для них не передали аргументи. Це робиться дуже простим та очевидним чином - достатньо вказати значення за замовчуванням безпосередньо на момент оголошення параметрів у підписі функції. У випадку такого запису, якщо для параметра не було передано значення аргументу, використовується значення за замовчуванням.

function count(countFrom = 0, countTo = 10, step = 1) {  
 console.log(`countFrom = ${countFrom}, countTo = ${countTo}, step = ${step}`);  
  
 for (let i = countFrom; i <= countTo; i += step) {  
 console.log(i);  
 }  
}  
  
count(1, 5); *// countFrom = 1, countTo = 5, step = 1*  
count(2); *// countFrom = 2, countTo = 10, step = 1*  
count(); *// countFrom = 0, countTo = 10, step = 1*

## Псевдомасив arguments[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/functions" \l "%D0%BF%D1%81%D0%B5%D0%B2%D0%B4%D0%BE%D0%BC%D0%B0%D1%81%D0%B8%D0%B2-arguments" \o "Direct link to heading)

Доступ до списку всіх **аргументів** можна отримати за допомогою спеціальної змінної arguments, яка доступна тільки всередині функції і зберігає всі аргументи у якості псевдомасиву.

**Псевдомасив** - колекція з властивістю length і можливістю звернутися до елементу за індексом, але відсутністю більшості методів для роботи з масивом.

Розглянемо приклад використання arguments у функції, яка множить будь-яку кількість аргументів:

function multiply() {  
 let total = 1;  
  
 for (const argument of arguments) {  
 total \*= argument;  
 }  
  
 return total;  
}  
  
console.log(multiply(1, 2, 3)); *// 6*  
console.log(multiply(1, 2, 3, 4)); *// 24*  
console.log(multiply(1, 2, 3, 4, 5)); *// 120*

### Перетворення псевдомасиву[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/functions" \l "%D0%BF%D0%B5%D1%80%D0%B5%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D0%BF%D1%81%D0%B5%D0%B2%D0%B4%D0%BE%D0%BC%D0%B0%D1%81%D0%B8%D0%B2%D1%83" \o "Direct link to heading)

Зазвичай псевдомасив необхідно перетворити у повноцінний масив, оскільки у псевдомасиву відсутні методи масиву, наприклад slice() або includes(). На практиці застосовують декілька основних способів.

Використовуючи метод Array.from(), який створить масив із псевдомасиву.

function fn() {  
 *// Змінна args буде містити повноцінний масив*  
 const args = Array.from(arguments);  
}

Використовуючи операцію ... (rest), вона дозволяє зібрати будь-яку кількість елементів, у нашому випадку аргументів, в масив, і зберегти його в змінну. Збираємо всі аргументи, використовуючи операцію rest безпосередньо в підписі функції.

function fn(...args) {  
 *// Змінна args буде містити повноцінний масив*  
}

ЦІКАВО

Операція rest детальніше розглядається далі на курсі, тут показаний один з її можливих варіантів застосування.

## Патерн «Раннє повернення»[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/functions#%D0%BF%D0%B0%D1%82%D0%B5%D1%80%D0%BD-%D1%80%D0%B0%D0%BD%D0%BD%D1%94-%D0%BF%D0%BE%D0%B2%D0%B5%D1%80%D0%BD%D0%B5%D0%BD%D0%BD%D1%8F)

Оператор if...else - це основний спосіб створення розгалужень. Проте, складні вкладені розгалуження роблять код заплутаним для розуміння.

Створимо функцію, яка обробляє зняття коштів з особистого рахунку в банку. Вона отримує суму для зняття і поточний баланс рахунку, після чого, залежно від умови, виконує необхідний блок коду.

function withdraw(amount, balance) {  
 if (amount === 0) {  
 console.log("Для проведення операції введіть суму більшу за нуль");  
 } else if (amount > balance) {  
 console.log("Недостатньо коштів на рахунку");  
 } else {  
 console.log("Операція зняття коштів проведена успішно");  
 }  
}  
  
withdraw(0, 300); *// "Для проведення операції введіть суму більшу за нуль"*  
withdraw(500, 300); *// "Недостатньо коштів на рахунку"*  
withdraw(100, 300); *// "Операція зняття коштів проведена успішно"*

Навіть у такому простому прикладі є група вкладених умовних операторів, серед яких не одразу можна зрозуміти логіку виконання коду.

У функції може бути більше одного оператора return. Головне пам'ятати, що виконання функції переривається, коли інтерпретатор зустрічає повернення, і увесь код після нього буде проігнорований в поточному виконанні функції.

**Патерн «Раннє повернення»** - це спосіб використовувати можливість дострокового повернення з функції за допомогою оператора return. Використовуючи цей прийом, ми отримуємо чистіший, плоскіший і зрозуміліший код, який простіше рефакторити.

Виділимо всі перевірки умов в окремі оператори if, після чого додамо код, що знаходиться в тілі else. В ідеальному випадку, повинен вийти плоский список умовних операторів, що йдуть один за одним, а в кінці - блок, який виконається тільки у тому випадку, якщо не виконається жоден if.

function withdraw(amount, balance) {  
 *// Якщо умова виконується, викликається console.log*  
 *// і вихід із функції. Код після тіла if не виконається.*  
 if (amount === 0) {  
 console.log("Для проведення операції введіть суму більшу за нуль");  
 return;  
 }  
  
 *// Якщо умова першого if не виконалась, його тіло пропускається*  
 *// та інтерпретатор доходе до другого if.*  
 *// Якщо умова виконується, викликається console.log і вихід із функції.*  
 *// Код, що знаходиться після тіла if, не виконається.*  
 if (amount > balance) {  
 console.log("Недостатньо коштів на рахунку");  
 return;  
 }  
  
 *// Якщо жоден із попередніх if не виконався,*  
 *// інтерпретатор доходить до цього коду і виконує його.*  
 console.log("Операція зняття коштів проведена");  
}  
  
withdraw(0, 300); *// "Для проведення операції введіть суму більшу за нуль"*  
withdraw(500, 300); *// "Недостатньо коштів на рахунку"*  
withdraw(100, 300); *// "Операція зняття коштів проведена"*

## Функціональний вираз[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/functions#%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%B8%D0%B9-%D0%B2%D0%B8%D1%80%D0%B0%D0%B7)

**Функціональний вираз (function expression)** - звичайне оголошення змінної, значенням якої буде функція. Альтернативний спосіб оголошення функції.

*// Оголошення функції (function declaration)*  
function multiply(x, y, z) {  
 console.log(`Результат множення дорівнює ${x \* y \* z}`);  
}  
  
*// Функціональний вираз (function expression)*  
const multiply = function (x, y, z) {  
 console.log(`Результат множення дорівнює ${x \* y \* z}`);  
};

Різниця в тому, що функціональний вираз не можна викликати до місця його створення, тільки після нього, тому що це буквально оголошення const змінної.

*// ❌ Помилка! Не працює виклик до оголошення*  
multiply(1, 2, 3);  
  
const multiply = function (x, y, z) {  
 console.log(`Результат множення дорівнює ${x \* y \* z}`);  
};  
  
*// ✅ Працює виклик після оголошення*  
multiply(4, 5, 6);

A оголошення функції можна викликати до місця її створення в коді.

*// ✅ Працює виклик до оголошення*  
multiply(1, 2, 3);  
  
function multiply(x, y, z) {  
 console.log(`Результат множення дорівнює ${x \* y \* z}`);  
}  
  
*// ✅ Працює виклик після оголошення*  
multiply(4, 5, 6);

# Стек вызовов

При вызове функции, внутри её тела могут вызываться другие функции, а в них другие и т. д. JavaScript однопоточный язык, то есть в одну единицу времени может выполняться только одна инструкция. Это значит, что уже вызванные функции, которые не закончили свое выполнение, должны ждать выполнения функций вызванных внутри себя, для того, чтобы продолжить свою работу.

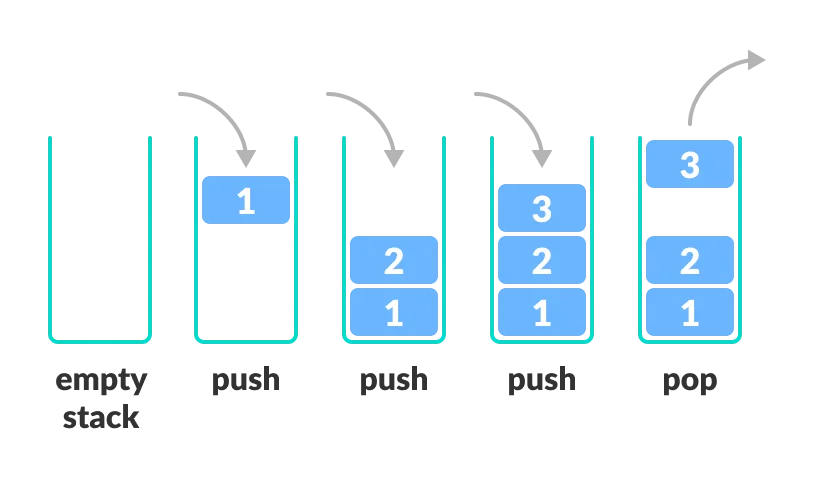
function fnA() {  
 console.log("Лог внуртри функции fnA до вызова fnB");  
 fnB();  
 console.log("Лог внуртри функции fnA после вызова fnB");  
}  
  
function fnB() {  
 console.log("Лог внутри функции fnB");  
}  
  
console.log("Лог перед вызовом fnA");  
fnA();  
console.log("Лог после вызова fnA");  
  
*// "Лог перед вызовом fnA"*  
*// "Лог внуртри функции fnA до вызова fnB"*  
*// "Лог внутри функции fnB"*  
*// "Лог внуртри функции fnA после вызова fnB"*  
*// "Лог после вызова fnA"*

Необходим механизм хранения списка функций, которые были вызваны, но еще не закончили свое выполнение и механизм управления порядком выполнения этих функций - и именно за это отвечает стек вызовов (call stack).

## Стек[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/callstack#%D1%81%D1%82%D0%B5%D0%BA)

**Стек** - структура данных, которая работает по принципу LIFO (Last-In-First-Out), то есть последним пришёл - первым вышел. Последнее, что добавляется на стек, будет удалено из него первым, значит можно добавить или удалить элементы только из верхушки стека.

Представьте стек как массив у которого есть только методы pop и push, то есть можно добавить или удалить только элемент в конце коллекции.



## Стек вызовов[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/callstack" \l "%D1%81%D1%82%D0%B5%D0%BA-%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2%D0%BE%D0%B2" \o "Direct link to heading)

**Стек вызовов (call stack)** - это механизм для отслеживания текущего местонахождения интерпретатора в коде, который вызывает несколько функций. Какая из функций выполняется на данный момент, какие функции вызываются изнутри выполняемой функции, какая будет вызвана следующей и т. д.

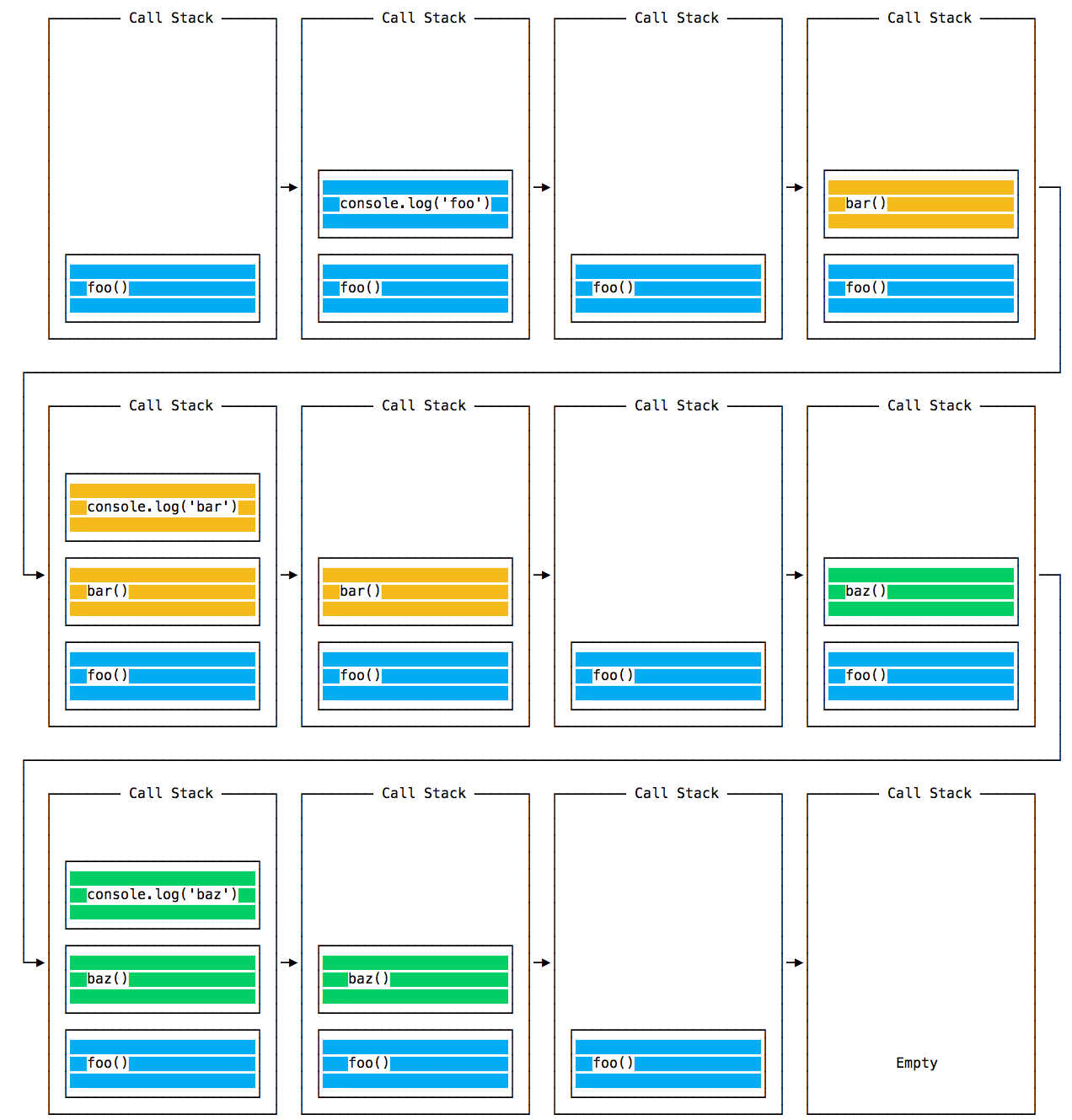
* Когда скрипт вызывает функцию, интерпретатор добавляет её в стек вызовов и начинает выполнение.
* Любые функции, вызванные выполняемой функцией, добавляются в стек вызовов и выполняются, как только происходит их вызов.
* Когда выполнение функции завершено, интерпретатор снимает её со стека вызовов и возобновляет выполнение кода с той точки, где остановился до этого. То есть начинает выполняться функция, запись которой лежит следующей на стеке.

ИНТЕРЕСНО

**Stack frame (кадр стека, запись стека)** - структура которая добавляется на стек при вызове функции. Хранит служебную информацию, например имя функции и номер строки, в которой произошел вызов.

function bar() {  
 console.log("bar");  
}  
  
function baz() {  
 console.log("baz");  
}  
  
function foo() {  
 console.log("foo");  
 bar();  
 baz();  
}  
  
foo();

При выполнении этого кода сначала вызывается foo(), затем внутри foo() вызывается bar(), а затем baz(). Вызовы console.log() так же учитываются, ведь это функция. На иллюстрации ниже пошагово изображен стек вызовов для примера.



## Переполнение стека вызовов[​](https://textbook.edu.goit.global/lms-it-generation-js/uk/docs/lesson-02/callstack" \l "%D0%BF%D0%B5%D1%80%D0%B5%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5-%D1%81%D1%82%D0%B5%D0%BA%D0%B0-%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2%D0%BE%D0%B2" \o "Direct link to heading)

Стек вызовов не безграничный, ему отводится конечный объекм памяти. Иногда в консоли можно увидеть ошибку "Uncaught RangeError: Maximum call stack size exceeded" - переполнение стека (stack overflow).

Это может произойти при неправильном использовании рекурсии или зацикливании вызовов функций, то есть, если идут бесконечные вызовы функций и результат не возвращается, то стек увеличивается. По достижению предела количества записей стека и возникнет такая ошибка и скрипт «падает».

## Scope chain search