1. **Які існують типи даних в JavaScript ?**

|  |  |  |
| --- | --- | --- |
| Type | typeof return value | Object wrapper |
| Null | "object" | N/A |
| Undefined | "undefined" | N/A |
| Boolean | "boolean" | Boolean |
| Number | "number" | Number |
| BigInt | "bigint" | BigInt |
| String | "string" | String |
| Symbol | "symbol" | Symbol |

1. **Що таке use strict і для чого він використовується?**

**What is "use strict"?**

"use strict" is a special statement that you can add to the beginning of your script or function to activate the strict mode. The strict mode is a feature of JavaScript that changes the behavior of some features and syntax, making them more consistent, predictable, and secure. For example, the strict mode prevents you from using undeclared variables, deleting properties or functions, or using reserved words as identifiers. If you try to do any of these things, the strict mode will throw an error and stop the execution.

**How to use "use strict"?**

To use the "use strict" directive, you simply need to write it as a string literal at the top of your script or function. For example:

"use strict";

// your code here

This will apply the strict mode to the entire script. If you want to use the strict mode only for a specific function, you can write it inside the function body. For example:

function myFunction() {

"use strict";

// your code here

}

This will apply the strict mode only to the function scope. You can also use the strict mode in modules or classes, which are implicitly in strict mode by default.

1. **Що таке this в JavaScript і як він працює?**

In JavaScript, the this keyword refers to an object.

The this keyword refers to different objects depending on how it is used:

* In an object method, this refers to the object.
* Alone, this refers to the global object.
* In a function, this refers to the global object.
* In a function, in strict mode, this is undefined.
* In an event, this refers to the element that received the event.
* Methods like call(), apply(), and bind() can refer this to any object.

1. **Як працюють методи call, bind, та apply? Наведіть приклади їх використання.**

Методи call, apply і bind в JavaScript використовуються для встановлення контексту this в функціях. Вони дозволяють викликати функцію з вказаним об'єктом як this.

Метод call викликає функцію одразу, передаючи this та аргументи через кому.

function greet(greeting, punctuation) {

console.log(`${greeting}, ${this.name}${punctuation}`);

}

const person = { name: "Іван" };

greet.call(person, "Привіт", "!");

// Виведе: Привіт, Іван!

Метод apply працює аналогічно call, але аргументи передаються у вигляді масиву.

greet.apply(person, ["Доброго дня", "."]); // Виведе: Доброго дня, Іван.

Практичний приклад apply — Math.max()

const numbers = [3, 7, 2, 9, 5];

const max = Math.max.apply(null, numbers);

console.log(max); // 9

Метод bind не викликає функцію одразу, а повертає нову функцію з прив'язаним this.

const boundGreet = greet.bind(person, "Привіт");

boundGreet("!!!");

// Виведе: Привіт, Іван!!!

const user = {

name: "Ольга",

sayHello() {

console.log(`Привіт, я ${this.name}`);

}

};

const say = user.sayHello;

say(); // undefined, бо `this` втрачається

const boundSay = user.sayHello.bind(user);

boundSay(); // Привіт, я Ольга

1. **Яка різниця між var, let та const?**

*var:*

* Область видимості: функціональна (видима у всій функції, де її оголосили).
* Підйом (hoisting): піднімається на початок своєї області, але має значення undefined.
* Можна перевизначити та повторно оголосити: так.

*let:*

* Область видимості: блокова (тільки в {})
* Підйом (hoisting): піднімається, але не ініціалізується (якщо звернутися до неї до оголошення, буде ReferenceError).
* Можна змінювати значення, але не можна повторно оголошувати в межах однієї області видимості.

*const:*

* Область видимості: блокова.
* Підйом (hoisting): аналогічно let – піднімається, але не ініціалізується.
* Не можна змінювати значення після оголошення.
* Об'єкти та масиви можна змінювати, але не можна перевизначити.

1. **Що таке підйом (hoisting) і як він працює?**

Підйом (hoisting) — це механізм JavaScript, при якому оголошення змінних та функцій "піднімаються" на початок своєї області видимості перед виконанням коду.

Проте підйом працює по-різному для var, let / const та функцій.

|  |  |  |  |
| --- | --- | --- | --- |
| Тип | Піднімається? | Ініціалізується? | Доступ до оголошення? |
| var | ✅ Так | ❌ undefined | ✅ Так (але undefined) |
| let | ✅ Так | ❌ Ні | ❌ ReferenceError |
| const | ✅ Так | ❌ Ні (обов'язково ініціалізувати) | ❌ ReferenceError |
| function declaration | ✅ Так | ✅ Так | ✅ Так |
| function expression | ✅ Так | ❌ Ні | ❌ ReferenceError |

1. **Що таке захоплення (capture) та спливання (bubbling) подій?**

Коли подія (наприклад, click) відбувається на елементі HTML, вона не просто виконується на цьому елементі, а проходить через два фази:

Фаза захоплення (capturing phase) – подія спочатку проходить від кореня документа (document) до цільового елемента.

Фаза спливання (bubbling phase) – подія потім підіймається вгору від цільового елемента до document.

**<div id="parent">**

**<button id="child">Натисни мене</button>**

**</div>**

**//Bubbling**

document.getElementById("parent").addEventListener("click", function() {

console.log("Клік по батьківському елементу (parent)");

});

document.getElementById("child").addEventListener("click", function() {

console.log("Клік по дочірньому елементу (child)");

});

**Клік на кнопку #child:**

Клік по дочірньому елементу (child)

Клік по батьківському елементу (parent)

**//Capture**

document.getElementById("parent").addEventListener("click", function() {

console.log("Клік по батьківському елементу (parent)");

}, true); // true = режим захоплення

document.getElementById("child").addEventListener("click", function() {

console.log("Клік по дочірньому елементу (child)");

});

**Клік на кнопку #child:**

Клік по батьківському елементу (parent)

Клік по дочірньому елементу (child)

**Якщо потрібно зупинити подальше спливання події, використовується event.stopPropagation().**

document.getElementById("child").addEventListener("click", function(event) {

console.log("Клік по дочірньому елементу (child)");

event.stopPropagation(); // Зупиняє спливання

});

document.getElementById("parent").addEventListener("click", function() {

console.log("Клік по батьківському елементу (parent)");

});

Клік по дочірньому елементу (child)

1. **Яка різниця між методами map та forEach?**

Метод forEach() виконує передану функцію для кожного елемента масиву, але не повертає новий масив.

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

const doubled = numbers.map(num => num \* 2);

console.log(doubled); // [2, 4, 6, 8, 10]

Метод map() **створює новий масив**, в якому кожен елемент — це результат виконання переданої функції.

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

const doubled = numbers.map(num => num \* 2);

console.log(doubled); // [2, 4, 6, 8, 10]

Помилка: використання forEach() замість map()

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

const doubled = numbers.forEach(num => num \* 2);

console.log(doubled); // ❌ undefined

1. **Як зробити глибоку копію об’єкта**Чим відрізняється поверхнева копія від глибокої?

Поверхнева копія (shallow copy) – копіює лише верхній рівень об'єкта. Вкладені об'єкти залишаються посиланнями на оригінальні.

Глибока копія (deep copy) – створює повністю незалежну копію всіх вкладених об'єктів.

**Глибока копія через structuredClone() (сучасний метод)**const original = {

name: "Alice",

details: { age: 25, city: "New York" }

};

const copy = structuredClone(original);

copy.details.age = 30;

console.log(original.details.age); // 25 (не змінюється)

console.log(copy.details.age); // 30  
  
✔ Працює для складних об'єктів (масиви, вкладені об'єкти, Map, Set, Date).

❌ Не працює у старих браузерах (підтримується в більшості сучасних).  
  
**Глибока копія через JSON.parse(JSON.stringify(obj)) (простий спосіб)**const original = {

name: "Alice",

details: { age: 25, city: "New York" }

};

const copy = JSON.parse(JSON.stringify(original));

copy.details.age = 30;

console.log(original.details.age); // 25 (не змінюється)

console.log(copy.details.age); // 30  
  
✔Простий і працює у всіх браузерах.

❌ Втратить функції, undefined, Symbol, Date, Map, Set.

**Глибока копія через бібліотеку lodash**const \_ = require('lodash');

const original = {

name: "Alice",

details: { age: 25, city: "New York" }

};

const copy = \_.cloneDeep(original);

copy.details.age = 30;

console.log(original.details.age); // 25 (не змінюється)

console.log(copy.details.age); // 30

**✔ Коректно копіює всі типи даних.  
❌ Потрібно підключати Lodash.  
  
Глибока копія через рекурсію (ручний спосіб)**function deepCopy(obj) {

if (obj === null || typeof obj !== "object") return obj;

if (Array.isArray(obj)) {

return obj.map(deepCopy);

}

const copy = {};

for (let key in obj) {

if (obj.hasOwnProperty(key)) {

copy[key] = deepCopy(obj[key]);

}

}

return copy;

}

const original = {

name: "Alice",

details: { age: 25, city: "New York" }

};

const copy = deepCopy(original);

copy.details.age = 30;

console.log(original.details.age); // 25 (не змінюється)

console.log(copy.details.age); // 30  
  
✔ Гнучкий метод, не втрачає дані, працює без бібліотек.

❌ Може бути повільним для дуже великих об'єктів.

1. **Що таке замикання (Closures) в JavaScript?**Замикання (closure) – це функція, яка "запам'ятовує" свою область видимості (scope) навіть після того, як зовнішня функція завершила виконання.

Тобто: якщо функція створена всередині іншої функції, вона може зберігати доступ до змінних зовнішньої функції навіть після її виклику.

Приклад 1: Базове замикання

function outer() {

let message = "Привіт із замикання!";

function inner() {

console.log(message); // Має доступ до змінної зовнішньої функції

}

return inner;

}

const myClosure = outer(); // `outer` виконується, але `inner` ще не викликано

myClosure(); // Виведе: "Привіт із замикання!"  
  
📌 Що тут відбувається?

outer() виконується та повертає внутрішню функцію inner.

inner() "запам’ятовує" змінну message, навіть якщо outer() вже завершила виконання.

myClosure() викликає inner(), і вона все ще має доступ до message.  
  
Замикання для збереження стану  
  
function createCounter() {

let count = 0; // Локальна змінна

return function() {

count++;

console.log(`Лічильник: ${count}`);

};

}

const counter = createCounter();

counter(); // Лічильник: 1

counter(); // Лічильник: 2

counter(); // Лічильник: 3

📌 Що відбувається?

Функція createCounter() повертає внутрішню функцію, яка збільшує count.

count зберігається в пам'яті, навіть після завершення createCounter().

Кожен виклик counter() оновлює count.  
  
Замикання для приховування даних (Приватні змінні)  
  
function createSecret(secret) {

return {

getSecret: function() {

return secret;

}

};

}

const mySecret = createSecret("Це секрет!");

console.log(mySecret.getSecret()); // "Це секрет!"

console.log(mySecret.secret); // undefined (немає прямого доступу)  
  
📌 Навіщо це потрібно?

secret є приватною змінною, оскільки вона доступна тільки через getSecret().

Це імітація інкапсуляції, оскільки немає способу змінити secret напряму.  
  
🔥 Де використовуються замикання?  
  
✅ **Обробники подій**function attachHandler(message) {

return function() {

console.log(message);

};

}

const button = document.querySelector("button");

button.addEventListener("click", attachHandler("Кнопку натиснуто!"));  
  
✅ **Модулі (імітована приватність)**const counterModule = (function() {

let count = 0;

return {

increment: () => ++count,

getCount: () => count

};

})();

console.log(counterModule.increment()); // 1

console.log(counterModule.increment()); // 2

console.log(counterModule.getCount()); // 2

1. **Що таке LocalStorage і SessionStorage? Який максимальний розмір LocalStorage?**

LocalStorage і SessionStorage – це два механізми веб-зберігання (Web Storage API), які дозволяють зберігати дані в браузері користувача. Вони працюють схоже на cookies, але мають більший обсяг пам’яті та не відправляються на сервер із кожним HTTP-запитом.

🛠 **Різниця між LocalStorage і SessionStorage**

|  |  |  |
| --- | --- | --- |
| Характеристика | LocalStorage | SessionStorage |
| Тривалість зберігання | Доки користувач не видалить дані вручну | До закриття вкладки браузера |
| Обсяг пам’яті | ~5-10MB на домен | ~5MB на домен |
| Доступність між вкладками | Так, спільне для всіх вкладок цього домену | Ні, доступне тільки у межах однієї вкладки |
| Відправлення на сервер | ❌ Ніколи | ❌ Ніколи |

**📌 Максимальний розмір LocalStorage**

Залежно від браузера, LocalStorage зазвичай має обмеження 5MB на домен (у деяких браузерах може бути до 10MB).

Якщо спробувати зберегти більше, виникне помилка QuotaExceededError.

**🛠 Як використовувати LocalStorage та SessionStorage?**

✅ LocalStorage – збереження та отримання даних  
// Зберегти дані

localStorage.setItem("username", "JohnDoe");

// Отримати дані

console.log(localStorage.getItem("username")); // "JohnDoe"

// Видалити конкретний елемент

localStorage.removeItem("username");

// Очистити все сховище

localStorage.clear();  
  
✅ SessionStorage – тимчасове збереження

// Зберегти дані

sessionStorage.setItem("sessionID", "123456");

// Отримати дані

console.log(sessionStorage.getItem("sessionID")); // "123456"

// Видалити дані

sessionStorage.removeItem("sessionID");

// Очистити все сховище

sessionStorage.clear();  
  
⚠ **Примітка**: Не зберігайте в LocalStorage **конфіденційні дані** (паролі, токени), оскільки їх можна отримати через JavaScript.

1. **Опишіть, за допомогою чого в JS реалізуються такі ООП-парадигми, як інкапсуляція, поліморфізм, абстракція?**

Promise – це об'єкт у JavaScript, який представляє завершення або невдачу асинхронної операції. Він використовується для роботи з асинхронним кодом замість callback-функцій, що допомагає уникнути callback hell (вкладених викликів).

**🛠 Як працює Promise?**

Проміс має три стани:

1. Pending (Очікування) – Операція ще не завершена.
2. Fulfilled (Виконано) – Операція успішна, повертає результат.
3. Rejected (Відхилено) – Операція завершилася помилкою.

**Створення промісу**

const myPromise = new Promise((resolve, reject) => {

let success = true; // Симуляція успіху або помилки

setTimeout(() => {

if (success) {

resolve("✅ Операція успішна!");

} else {

reject("❌ Сталася помилка!");

}

}, 2000);

});

Використовуємо resolve() для успішного виконання та reject() у разі помилки.  
  
**Обробка промісу**myPromise

.then(result => console.log(result)) // Виконається, якщо проміс успішний

.catch(error => console.log(error)) // Виконається, якщо сталася помилка

.finally(() => console.log("✅ Операція завершена!")); // Завжди виконається  
  
📝 **.then()** – отримує значення з resolve().  
📝 **.catch()** – отримує значення з reject().  
📝 **.finally()** – виконується в будь-якому випадку.  
 **Ланцюжок (Chaining) промісів**function asyncTask() {

return new Promise((resolve) => {

setTimeout(() => resolve("Крок 1"), 1000);

});

}

asyncTask()

.then(result => {

console.log(result);

return "Крок 2"; // Передає далі

})

.then(result => {

console.log(result);

return "Крок 3";

})

.then(result => console.log(result))

.catch(error => console.log("Помилка:", error));  
  
Кожен .then() отримує результат попереднього .then() і передає далі.  
  
✅ **Promise.all()** – виконує всі проміси і повертає масив результатів.  
Promise.all([

Promise.resolve("🏆 Перший"),

new Promise(res => setTimeout(() => res("🥈 Другий"), 2000)),

new Promise(res => setTimeout(() => res("🥉 Третій"), 1000))

]).then(console.log);

// ["🏆 Перший", "🥉 Третій", "🥈 Другий"]  
✅ **Promise.race()** – повертає перший виконаний проміс.  
Promise.race([

new Promise(res => setTimeout(() => res("⏱ Швидкий"), 1000)),

new Promise(res => setTimeout(() => res("🐌 Повільний"), 3000))

]).then(console.log);

// "⏱ Швидкий"  
✅ **Promise.allSettled()** – чекає на всі проміси, повертає масив з їх статусами.  
Promise.allSettled([

Promise.resolve("✔️ Виконано"),

Promise.reject("❌ Помилка"),

]).then(console.log);

1. **Як працюють async та await?**

async та await – це синтаксичний цукор над Promise, який робить асинхронний код більш читабельним і зручним.

🔹 async перетворює функцію на проміс

🔹 await дозволяє зупинити виконання, поки проміс не завершиться  
  
**Приклад: без async/await (проміси)**function fetchData() {

return new Promise(resolve => {

setTimeout(() => resolve("📦 Дані отримані!"), 2000);

});

}

fetchData().then(data => console.log(data));

// Через 2 секунди: "📦 Дані отримані!"

**Те саме, але з async/await**  
async function getData() {

const data = await fetchData();

console.log(data);

}

getData();

// Через 2 секунди: "📦 Дані отримані!"  
  
  
**Обробка помилок з try...catch**  
async function getUser() {

try {

let response = await fetch("https://jsonplaceholder.typicode.com/users/1");

let user = await response.json();

console.log(user);

} catch (error) {

console.error("❌ Помилка:", error);

}

}

getUser();  
  
**Виконання кількох await послідовно (НЕЕФЕКТИВНО)**async function loadData() {

let data1 = await fetchData();

let data2 = await fetchData();

console.log(data1, data2);

}  
  
**Паралельне виконання await (ефективніше)**async function loadAll() {

let [data1, data2] = await Promise.all([fetchData(), fetchData()]);

console.log(data1, data2);

}

1. **Яка різниця між мікрозадачами (microtasks) та макрозадачами (macrotasks) в JavaScript?**JavaScript використовує Event Loop для обробки асинхронного коду.

Усі асинхронні операції розподіляються між мікрозадачами (Microtasks) та макрозадачами (Macrotasks), які виконуються у різних чергах.  
  
**🏎 1. Мікрозадачі (Microtasks)**

✅ Виконуються після кожної синхронної задачі і перед макрозадачами.

✅ Включають:

* Promise.then() / Promise.catch() / Promise.finally()
* queueMicrotask()
* MutationObserver

console.log("🌍 Початок");

Promise.resolve().then(() => console.log("🔹 Мікрозадача 1"));

Promise.resolve().then(() => console.log("🔹 Мікрозадача 2"));

console.log("🚀 Кінець");

**Результат:**  
🌍 Початок

🚀 Кінець

🔹 Мікрозадача 1

🔹 Мікрозадача 2

**🐢 2. Макрозадачі (Macrotasks)**

✅ Виконуються після усіх мікрозадач.

✅ Включають:

* setTimeout() / setInterval()
* setImmediate() (тільки Node.js)
* requestAnimationFrame()
* I/O операції, наприклад, fetch()

console.log("🌍 Початок");

setTimeout(() => console.log("🛑 Макрозадача 1"), 0);

setTimeout(() => console.log("🛑 Макрозадача 2"), 0);

Promise.resolve().then(() => console.log("🔹 Мікрозадача"));

console.log("🚀 Кінець");

**Результат:**  
🌍 Початок

🚀 Кінець

🔹 Мікрозадача

🛑 Макрозадача 1

🛑 Макрозадача 2  
  
  
**🏁 Як працює Event Loop?**

1️ Виконуються синхронні задачі (основний код).

2️ Виконується черга мікрозадач (усі Promise.then() та queueMicrotask()).

3️ Виконується перша макрозадача (наприклад, setTimeout()).

4️ Повернення до кроку 2 та повторення циклу.

1. **Яка різниця між функціональними виразами (function expressions) та оголошеннями функцій (function declarations)?  
     
   📝 1. Оголошення функцій (Function Declaration)**

✅ Має ім'я

✅ Піднімається (hoisting)

✅ Можна викликати до визначення

hello(); // ✅ Працює!

function hello() {

console.log("Привіт! 👋");

}

hello(); // ✅ Працює ще раз!  
  
**📌 Оголошені функції піднімаються (hoisting), тому їх можна викликати перед визначенням.  
  
📝 2. Функціональні вирази (Function Expression)**

✅ Зберігається в змінній

✅ Не піднімається (hoisting)

✅ Не можна викликати до визначення  
  
hello(); // ❌ Помилка! Cannot access 'hello' before initialization

const hello = function() {

console.log("Привіт! 👋");

};

hello(); // ✅ Працює після визначення!  
  
📌 **Функціональні вирази не піднімаються, тому їх не можна викликати до визначення.  
  
🎯 Коли використовувати що?**

Function Declaration добре підходить для глобальних функцій, які потрібні в усьому коді.

Function Expression краще використовувати, якщо функцію потрібно передати як аргумент або зберігати у змінній.

1. **Що таке event loop і як він працює?**

Концепція циклу подій дуже проста. Існує нескінченний цикл, в якому рушій JavaScript очікує завдання, виконує їх, а потім переходить в режим очікування нових завдань.

Загальний алгоритм рушія:

1.Поки є завдання:

* виконати їх, починаючи з найстарішого.

2.Очікувати поки завдання не з’явиться, потім перейти до пункту 1.

Це формалізація того, що ми бачимо, гортаючи вебсторінку. Рушій JavaScript більшість часу не робить нічого, він працює лише коли спрацьовує скрипт, обробник подій чи подія.

Приклади завдань:

* Коли завантажується зовнішній скрипт <script src="...">, тоді завдання полягає в виконанні цього скрипта.
* Коли користувач рухає мишкою, тоді завдання згенерувати подію mousemove і виконати її обробники.
* Коли пройде час, запрограмований в setTimeout, тоді завдання запустити його колбек.
* …і так далі.



Може трапитись так, що завдання приходить тоді, коли рушій вже зайнятий, тоді це завдання стає в чергу. Чергу з таких завдань називають “чергою макрозавдань”

Ще декілька деталей:

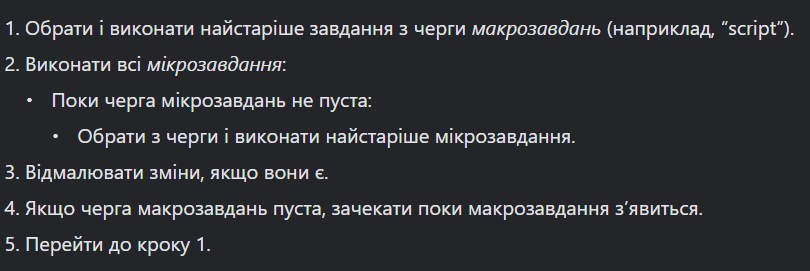
Рендеринг ніколи не відбувається поки рушій виконує завдання. Не має значення наскільки довго виконується завдання. Зміни в DOM будуть відмальовані лише після завершення завдання.

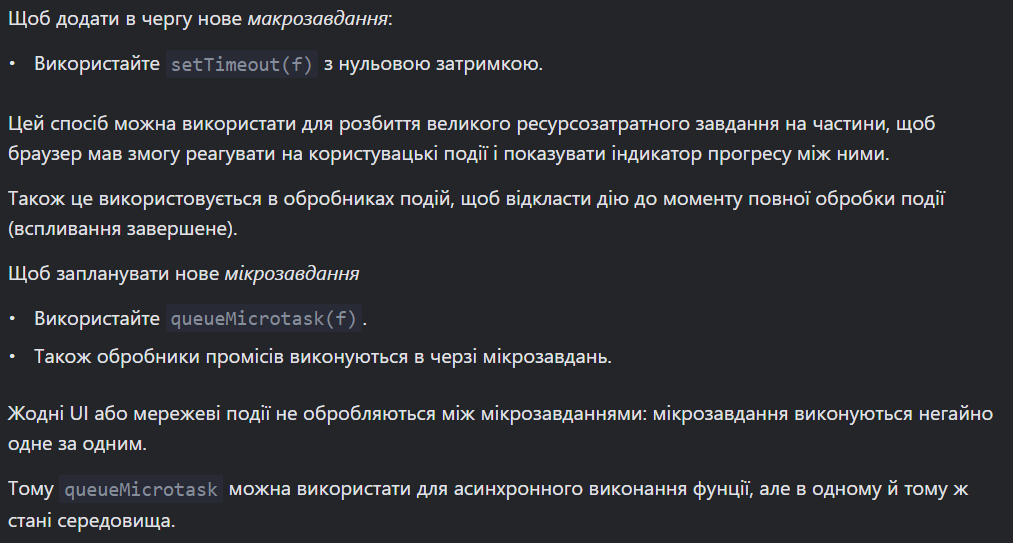
Якщо виконання завдання займає надто багато часу, браузер не зможе виконувати інші завдання, наприклад, обробляти користувацькі події. Тож після недовгого часу “зависання” з’явиться оповіщення “Сторінка не відповідає” і пропозиція вбити процес виконання завдання разом з цілою сторінкою. Таке трапляється коли код містить багато складних обрахунків або виникає програмна помилка, що створює нескінченний цикл.

Мікрозавдання приходять лише з нашого коду. Їх зазвичай створюють проміси: виконання обробника .then/catch/finally стає мікрозавданням. Мікрозавдання також використовуються “під капотом” await, так як це форма обробки проміса.

Також існує спеціальна функція queueMicrotask(func), яка ставить func в чергу мікрозавдань.

Одразу після кожного макрозавдання, рушій виконує всі завдання з черги мікрозавдань перед тим як виконати якесь макрозавдання чи рендеринг чи виконати щось іще.





1. **Що таке генератори функцій (function generators) та як вони використовуються?**

Звичайні функції повертають лише одне, єдине значення (або нічого).

Генератори можуть повертати (“yield”) кілька значень, одне за одним, на вимогу. Вони чудово працюють з об’єктами, що перебираються, дозволяючи легко створювати потоки даних.

Щоб створити генератор, нам потрібна спеціальна синтаксична конструкція: function\*, так звана “функція-генератор”.

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

Функції-генератори поводяться інакше, ніж звичайні. Коли така функція викликається, вона не запускає свій код. Замість цього вона повертає спеціальний об’єкт, який називається «об’єкт-генератор», щоб керувати її виконанням.

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

// "функція-генератор" створює "об’єкт-генератор"

let generator = generateSequence();

alert(generator); // [object Generator]

Основним методом генератора є next(). При виклику він запускає виконання коду до найближчого оператора yield <value> (value можна опустити, тоді воно є undefined). Потім виконання функції призупиняється, а отримане value повертається до зовнішнього коду.

Результатом next() завжди є об’єкт з двома властивостями:

value: отримане значення.

done: true, якщо код функції закінчився, інакше false.

**генератори є об’єктами, що перебираються.**

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

let generator = generateSequence();

for(let value of generator) {

alert(value); // 1, потім 2

}

Виглядає набагато приємніше, ніж виклик .next().value, чи не так?

…Але зверніть увагу: у прикладі вище показано 1, потім 2, і це все. Значення 3 не показується!

Це тому, що перебір через for..of ігнорує останнє value, коли done: true. Отже, якщо ми хочемо, щоб усі результати відображалися через for..of, то повинні повертати їх через yield

**Композиція генераторів – це особливість генераторів, що дозволяє прозоро “вбудовувати” генератори один в одного.**

function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) yield i;

}

Тепер ми хотіли б повторно використати його для створення складнішої послідовності:

* спочатку цифри 0..9 (з кодами символів 48…57),
* за якими йдуть великі літери алфавіту A..Z (коди символів 65…90)
* за якими йдуть малі літери алфавіту a..z (коди символів 97…122)

Ми можемо використовувати цю послідовність, наприклад створювати паролі, вибираючи з неї символи (можна також додати символи пунктуації), але давайте спочатку згенеруємо її.

У звичайній функції, щоб об’єднати результати кількох інших функцій, ми викликаємо їх, зберігаємо результати, а потім об’єднуємо в кінці.

Для генераторів існує спеціальний синтаксис yield\* для “вбудовування” (компонування) одного генератора в інший.

function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) yield i;

}

function\* generatePasswordCodes() {

// 0..9

yield\* generateSequence(48, 57);

// A..Z

yield\* generateSequence(65, 90);

// a..z

yield\* generateSequence(97, 122);

}

let str = '';

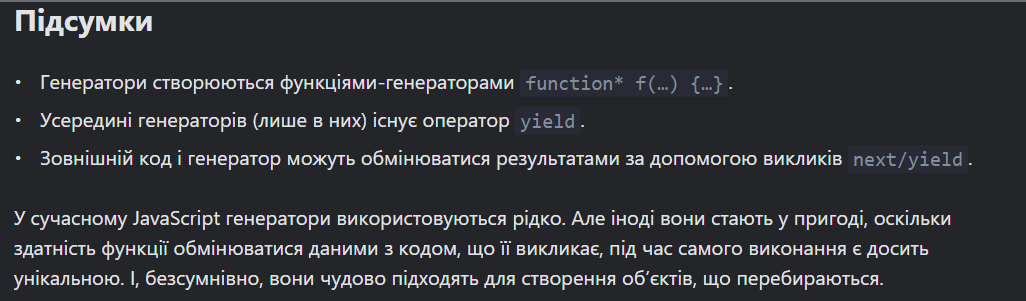
for(let code of generatePasswordCodes()) {

str += String.fromCharCode(code);

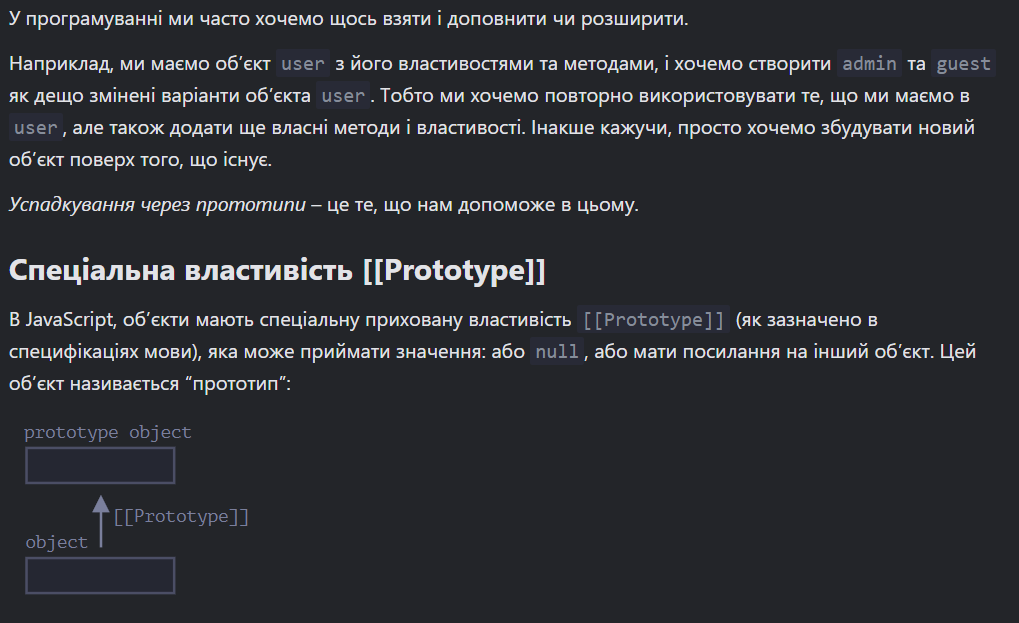
}

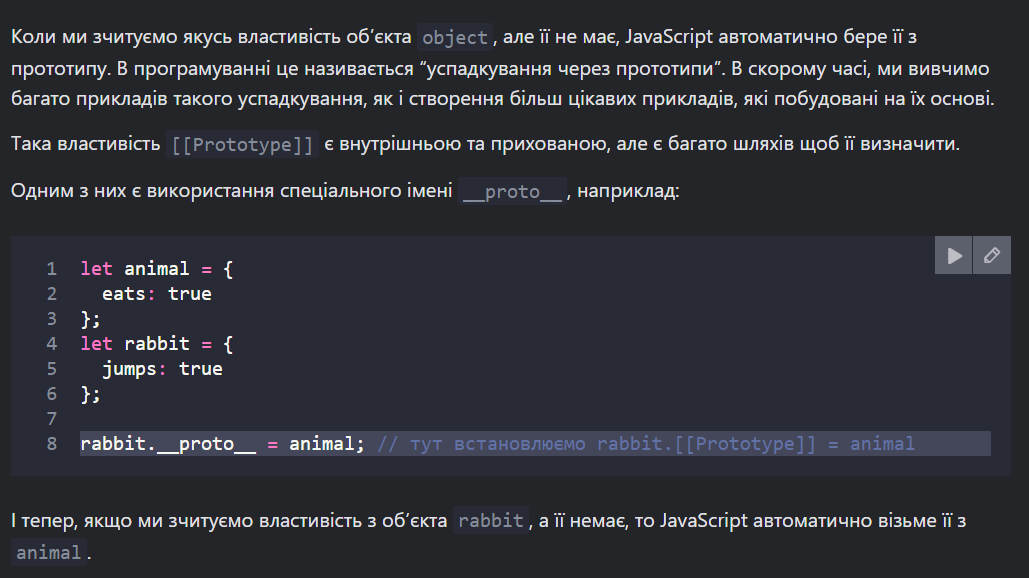
alert(str); // 0..9A..Za..z

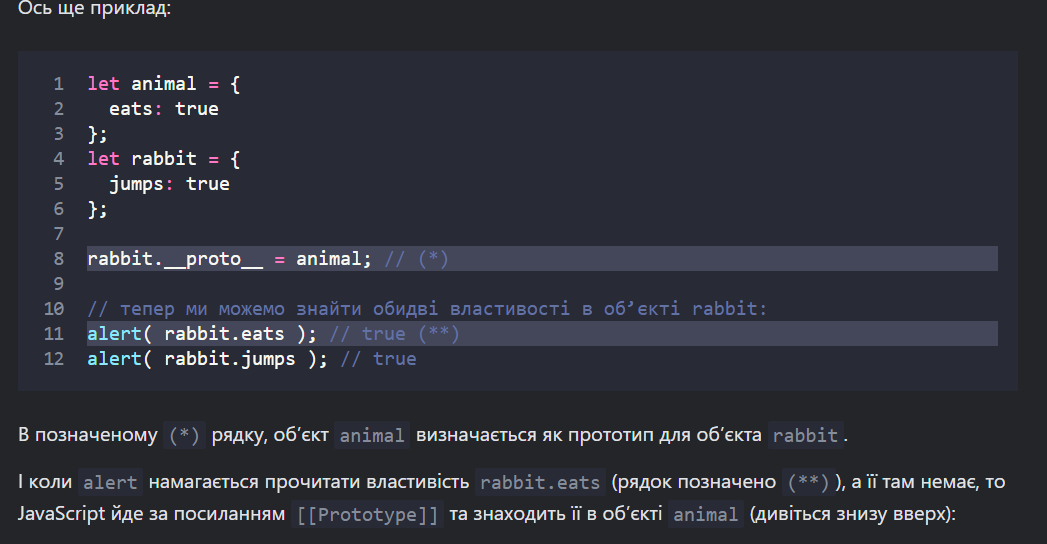
Директива yield\* делегує виконання іншому генератору. Цей термін означає, що yield\* gen виконує ітерацію над генератором gen і прозоро передає його вихід назовні. Ніби значення були отримані зовнішнім генератором.

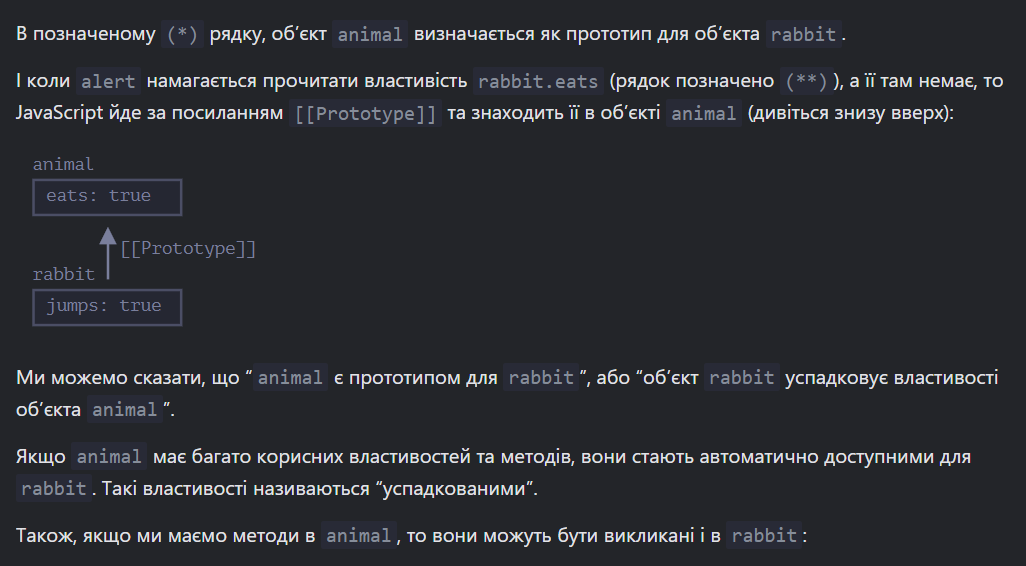


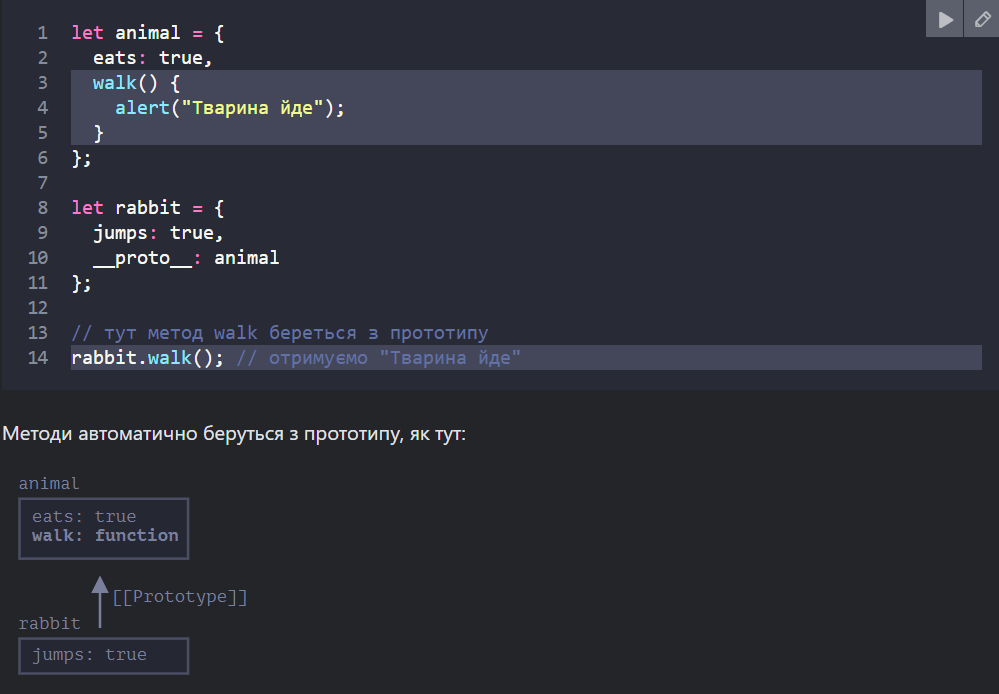
1. **Що таке proto та prototype? Як вони взаємодіють між собою?**

****

****

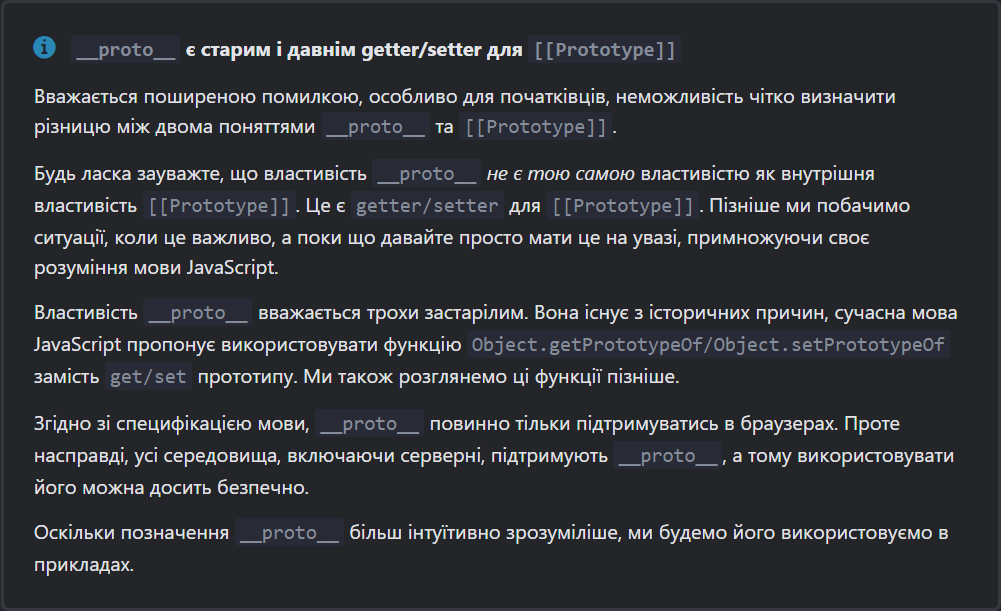
****

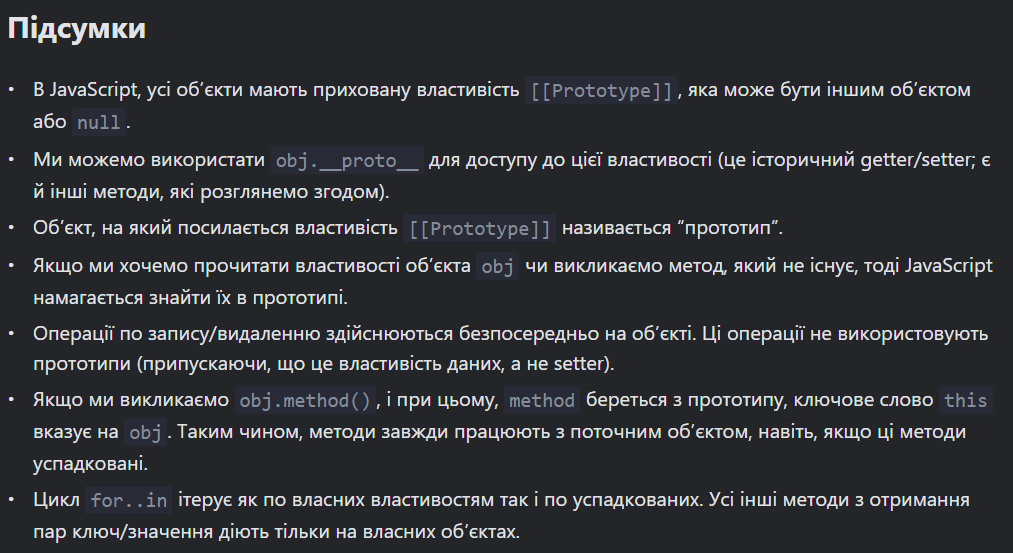
****

****

**Існує два обмеження:**

* **Посилання через прототипи не може бути замкнено в кільце. JavaScript видасть помилку, якщо ми визначемо \_\_proto\_\_ в ланцюжку прототипів і замкнем його в кільце.**
* **Значення \_\_proto\_\_ може бути, або посиланням на об’єкт, або null. Інші типи значень – ігноруються.**

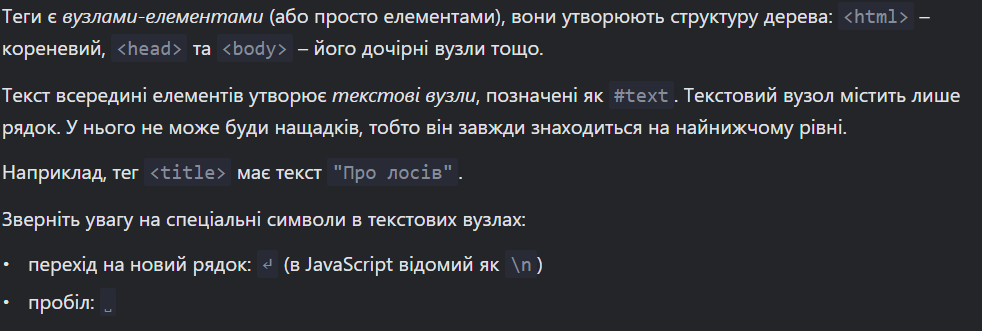
****

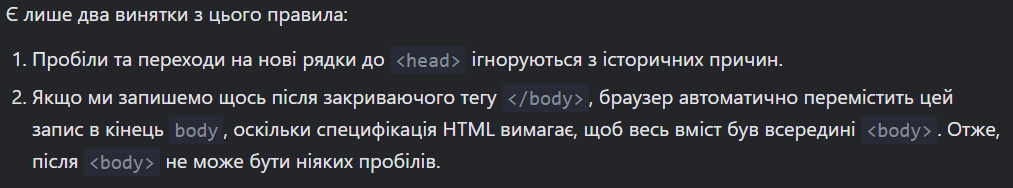
****

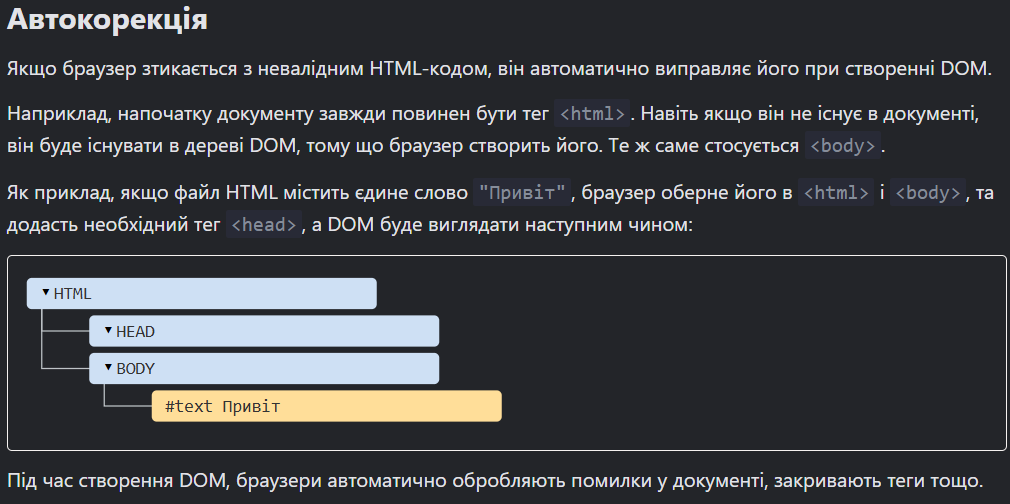
1. **Що таке Document Object Model (DOM) та як працює маніпуляція DOM?**

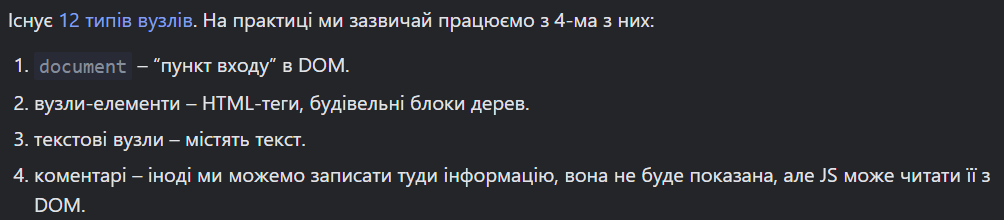
Відповідно до об’єктної моделі документа (“Document Object Model”, DOM), кожен HTML-тег є об’єктом. Вкладені теги є “дітьми” батьківського елементу. Текст, який знаходиться всередині тегу, також є об’єктом.

Всі ці об’єкти доступні за допомогою JavaScript, і ми можемо використовувати їх для зміни сторінки.

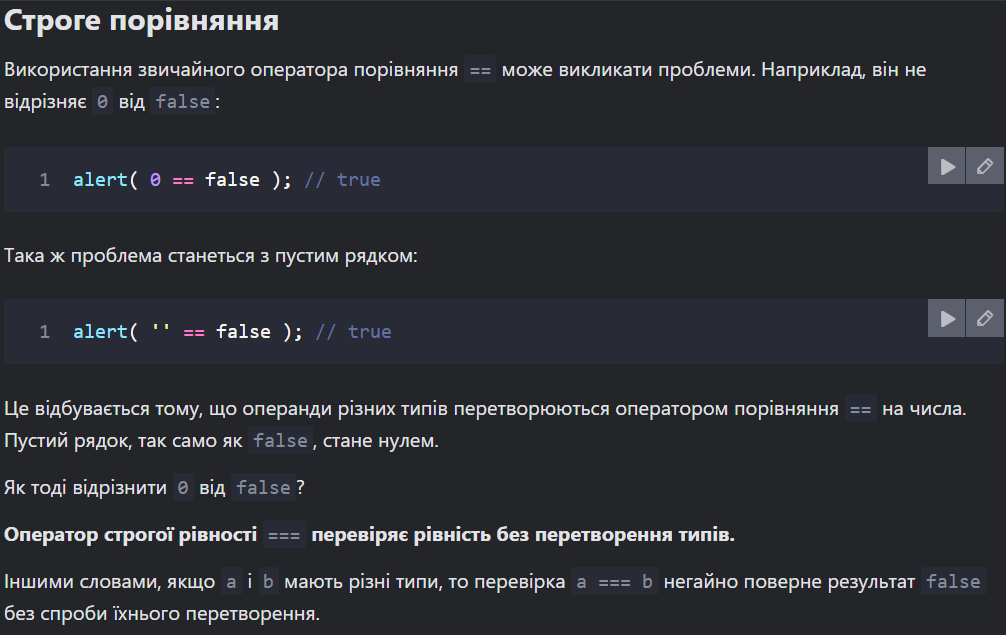




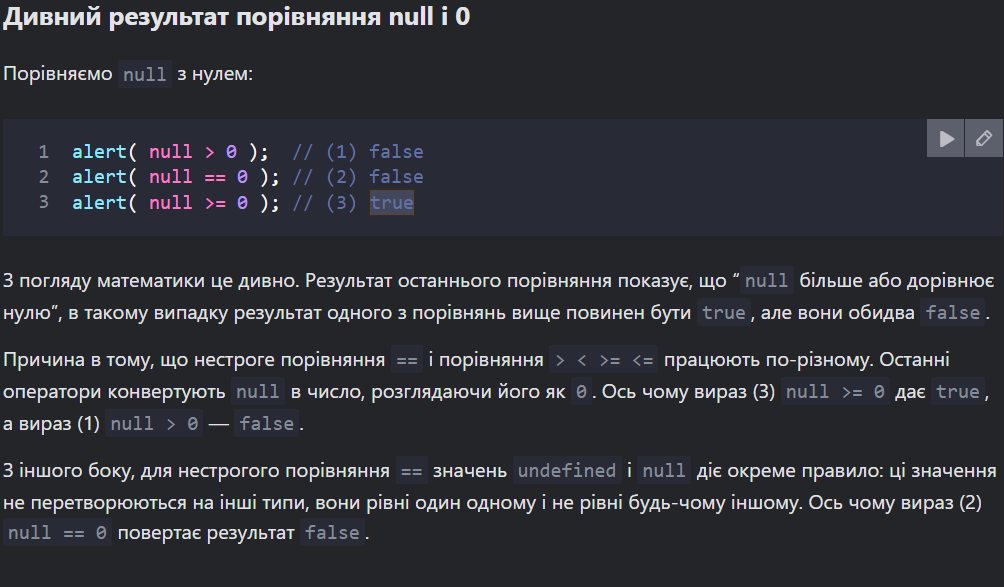


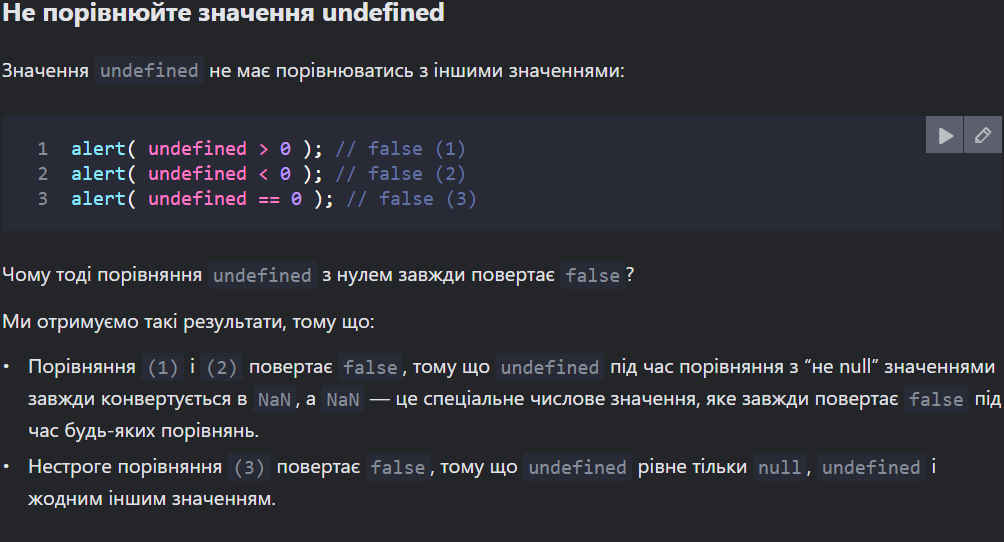
****

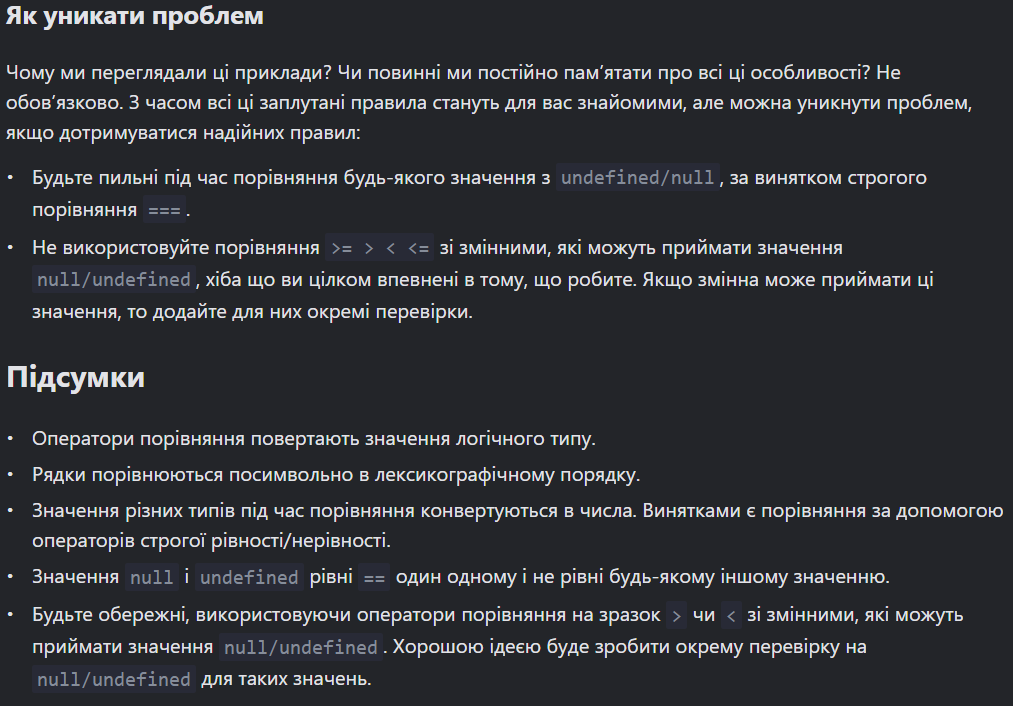
1. **У чому різниця між операторами == та ===?**

****

****

****

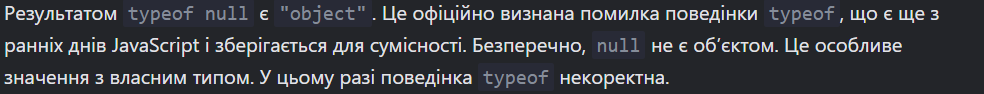
****

****

1. **Як перевірити тип даних в JavaScript?**

Перевірити тип даних можна за допомогою оператора typeof

1. **Чому typeof null повертає object?**

****

1. **Яка різниця між декларативним та імперативним підходами в програмуванні на JavaScript?**

**Імперативний підхід**

Імперативний стиль програмування фокусується на покроковому визначенні того, як саме потрібно досягти певного результату. Він описує алгоритм виконання дій, а не просто кінцевий стан.

const numbers = [1, 2, 3, 4, 5, 6];

const evenNumbers = [];

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

if (numbers[i] % 2 === 0) {

evenNumbers.push(numbers[i]);

}

}

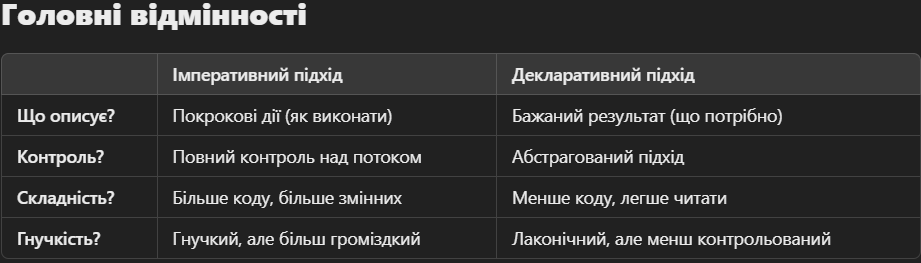
console.log(evenNumbers); // [2, 4, 6]  
  
**Декларативний підхід**

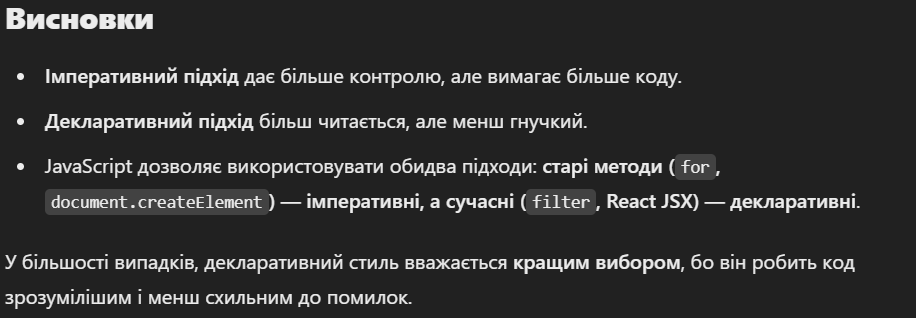
Декларативний стиль програмування фокусується на тому, що потрібно зробити, а не як це зробити. Код виглядає більш "чистим", тому що використовуються вбудовані методи мови, які приховують деталі реалізації.

const numbers = [1, 2, 3, 4, 5, 6];

const evenNumbers = numbers.filter(n => n % 2 === 0);

console.log(evenNumbers); // [2, 4, 6]





1. **Як працюють подієві делегати (event delegation)?**

Спливання та перехоплення дозволяють нам реалізувати один з найпотужніших шаблонів обробки подій під назвою делегування подій.

Ідея в тому, що якщо у нас є багато елементів, які обробляються подібним чином, то замість того, щоб призначати обробник кожному з них, ми ставимо один обробник на їхнього спільного предка.

У обробнику ми отримуємо event.target, щоб побачити, де насправді сталася подія і обробити її.  
  
<table>

<tr>

<th colspan="3">Таблиця <em>Багуа</em>: Напрямок, Елемент, Колір, Значення</th>

</tr>

<tr>

<td class="nw"><strong>Північний захід</strong><br>Метал<br>Срібний<br>Старійшини</td>

<td class="n">...</td>

<td class="ne">...</td>

</tr>

<tr>...Ще 2 подібні рядки...</tr>

<tr>...Ще 2 подібні рядки...</tr>

</table>

Таблиця має лише 9 клітинок, але їх може бути 99 або 9999, не має значення.

Наше завдання – виділити клітинку <td> при кліці на неї.

Замість того, щоб призначати обробник onclick для кожного <td> (їх може бути багато) – ми налаштуємо “універсальний” обробник для елемента <table>.

Він використовуватиме event.target, щоб отримати елемент, на який клікнули, і виділити його.

Код буде виглядати наступним чином:  
  
let selectedTd;

table.onclick = function(event) {

let target = event.target; // де відбувся клік?

if (target.tagName != 'TD') return; // не на TD? Тоді нас не цікавить

highlight(target); // виділити TD

};

function highlight(td) {

if (selectedTd) { // видалити наявне виділення, якщо таке є

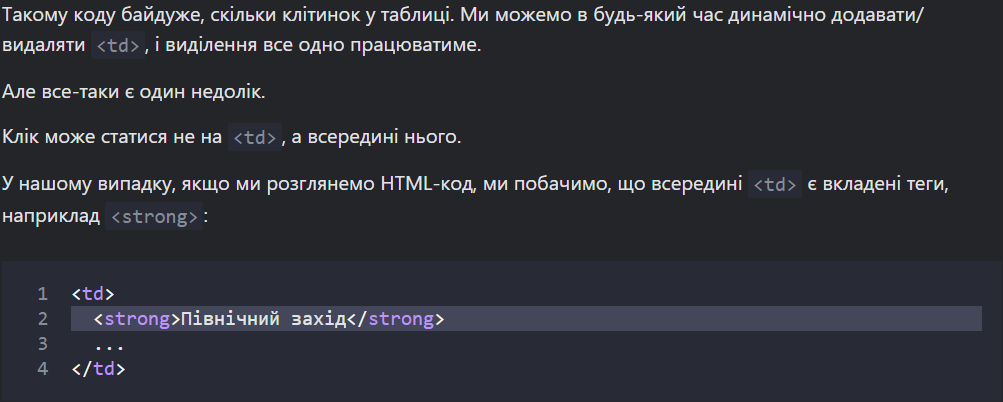
selectedTd.classList.remove('highlight');

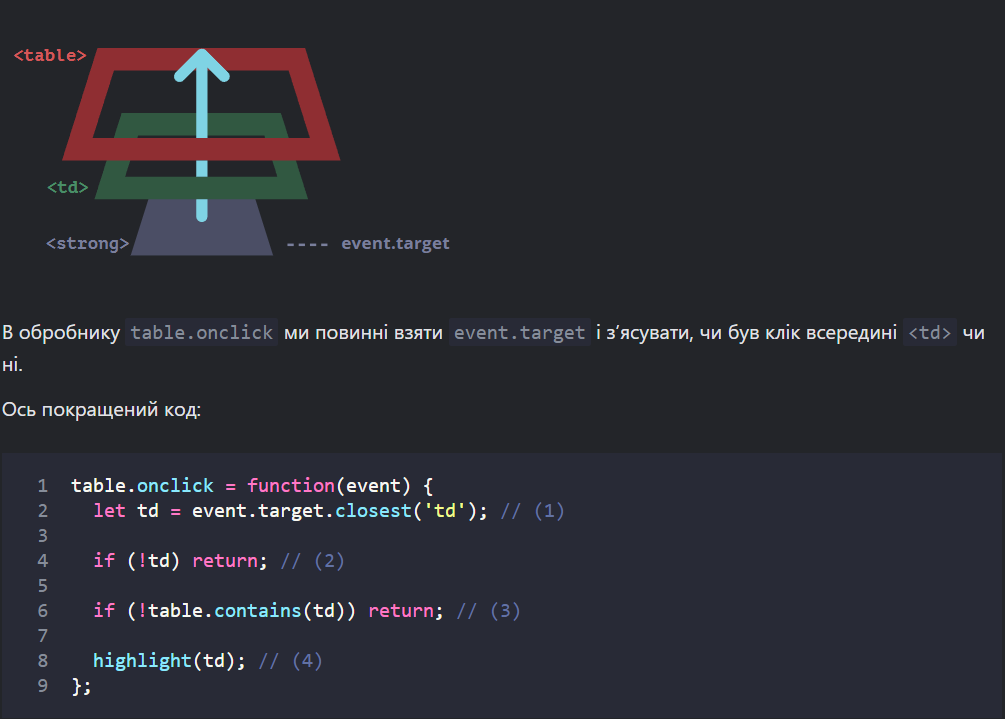
}

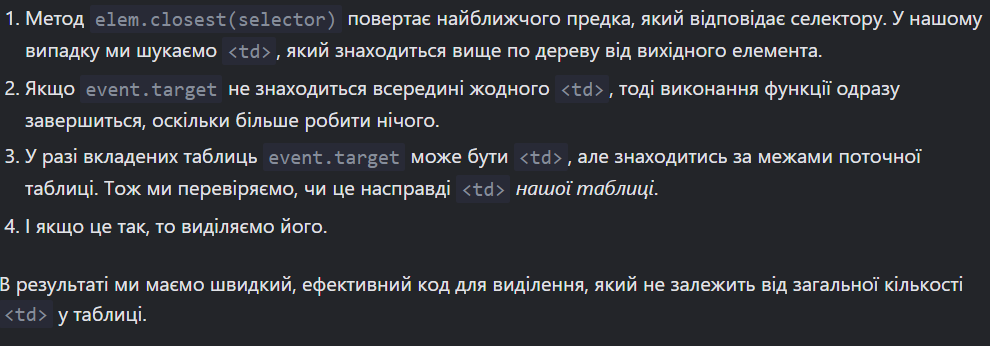
selectedTd = td;

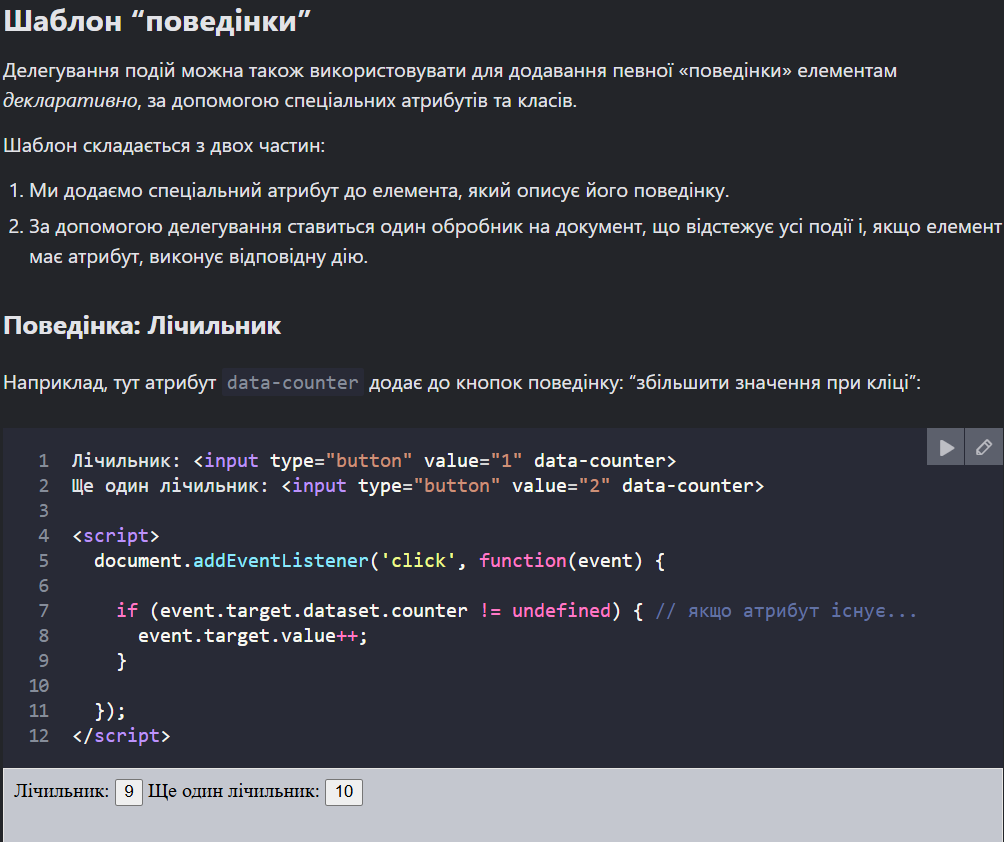
selectedTd.classList.add('highlight'); // виділити новий td

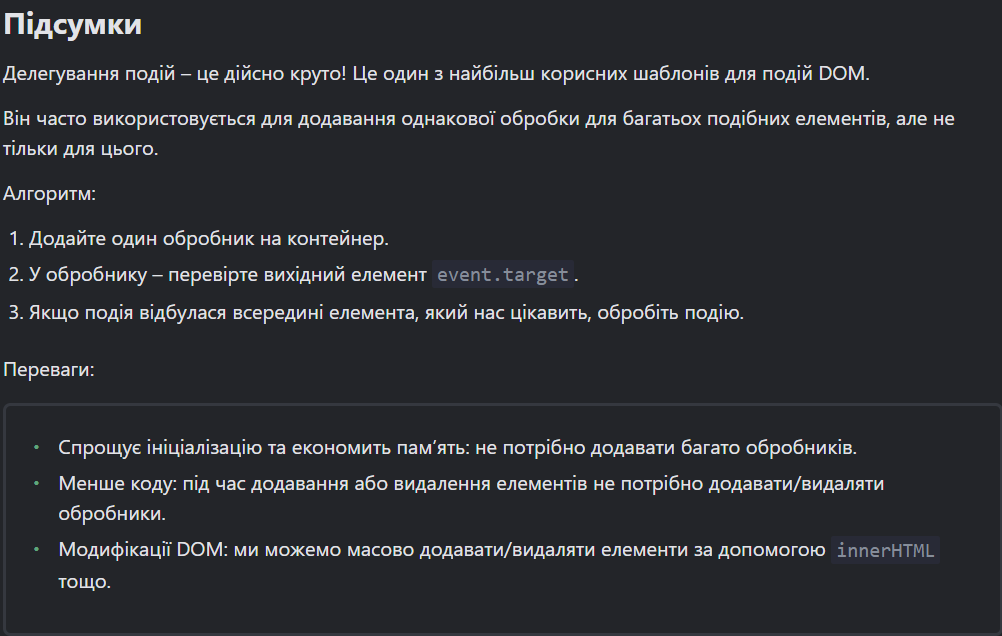
}











1. **Що таке захист від кодування (debouncing) та обмеження (throttling)?**

Throttling and debouncing are techniques used to control the rate at which a function is invoked, particularly in scenarios where frequent events (like user interactions) can lead to excessive calls and potentially impact performance.

**Throttling**

Throttling limits the number of times a function is executed over a certain time period. It ensures that the function is called at a controlled, steady rate, regardless of how frequently the event triggering the function occurs.

**Debouncing**

Debouncing is a technique that delays the execution of a function until a certain amount of time has passed since the last event's occurrence. It's useful when you want the function to be called after a period of inactivity, and it prevents rapid consecutive calls.

Still confused? 🤔 Here's a breakdown of the key differences between throttling and debouncing.

**1. Execution Behavior**

**Throttling:** In throttling, the function is executed at a fixed interval. Even if the triggering event occurs more frequently, the function is invoked according to the defined interval.

**Debouncing:** In debouncing, the function is only executed after a specific delay since the last event's occurrence. If new events occur within the delay period, the timer is reset, and the function execution is further delayed.

**2. Use Cases**

**Throttling:** Throttling is suitable for scenarios where you want to limit the frequency of function calls, like handling scroll events or resizing events. It helps avoid overloading the system with frequent updates.

**Debouncing:** Debouncing is ideal when you want to wait for a pause in the events before triggering a function. This is useful for situations like search suggestions, where you want to wait for the user to finish typing before fetching suggestions.

**3. Implementation**

**Throttling:** Throttling typically involves setting a fixed interval between function calls using timers or timestamps to track the last invocation time.

**Debouncing:** Debouncing involves starting a timer when an event occurs and resetting the timer whenever a new event occurs within the delay period. The function is executed when the timer expires after the last event.

**function throttle(func, delay) {**

**let lastCall = 0;**

**return function (...args) {**

**const now = new Date().getTime();**

**if (now - lastCall >= delay) {**

**func(...args);**

**lastCall = now;**

**}**

**};**

**}**

**const throttledScrollHandler = throttle(() => {**

**console.log("Loading more content...");**

**}, 1000);**

**window.addEventListener("scroll", throttledScrollHandler);**

In this example, the throttle function wraps the original function (scrollHandler) and ensures that it's called at most once every 1000 milliseconds (1 second) no matter how quickly the user scrolls.

**function debounce(func, delay) {**

**let timeoutId;**

**return function (...args) {**

**clearTimeout(timeoutId);**

**timeoutId = setTimeout(() => {**

**func(...args);**

**}, delay);**

**};**

**}**

**const debounceSearch = debounce((query) => {**

**console.log(`Searching for: ${query}`);**

**// Make API call with the search query**

**}, 300);**

**const searchInput = document.getElementById("search-input");**

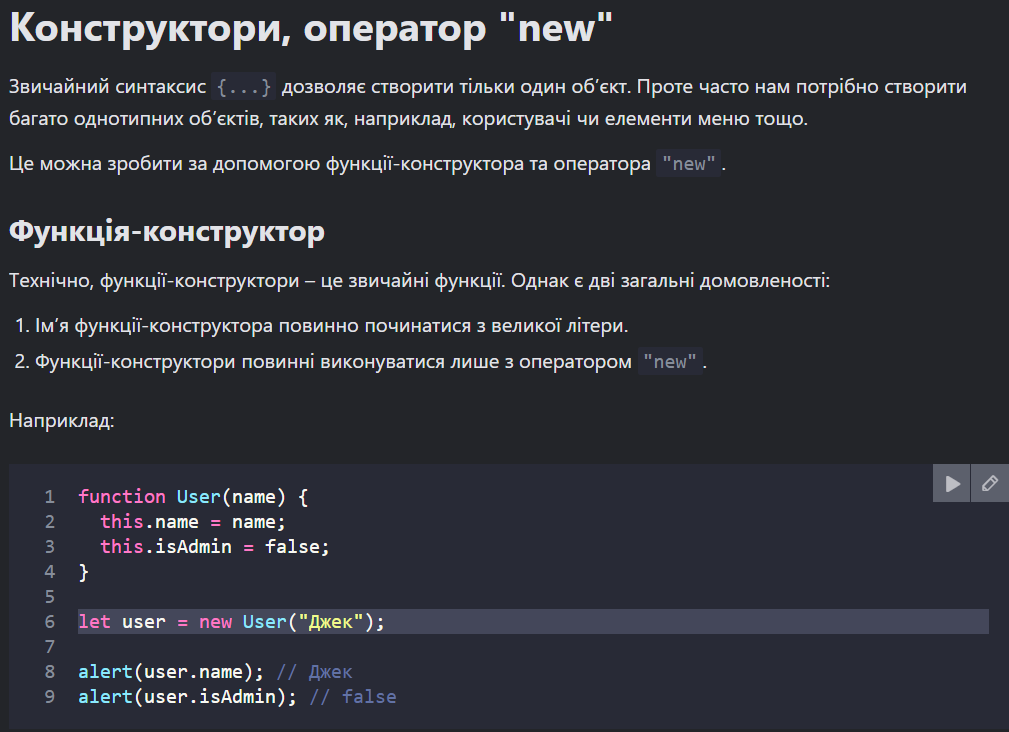
**searchInput.addEventListener("input", (event) => {**

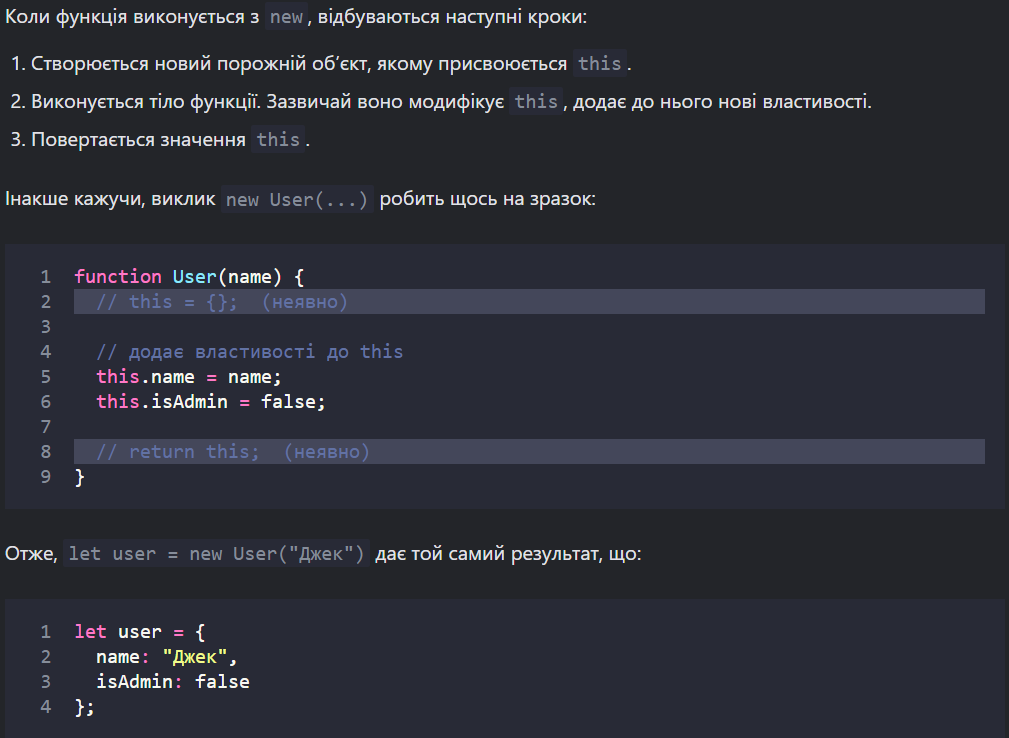
**debounceSearch(event.target.value);**

**});**

In this example, the debounce function ensures that the API call is made 300 milliseconds after the user stops typing. If the user continues typing, the timer is reset, preventing the API call from being triggered too frequently.

1. **Як працює механізм new в JavaScript?**

****



1. **Що таке IIFE (Immediately Invoked Function Expression) та як воно використовується?**

IIFE (Immediately Invoked Function Expression) — це анонімна функція, яка виконується відразу після того, як вона була визначена. Основною причиною використання IIFE є створення ізольованого простору (scope), що запобігає потраплянню змінних або функцій у глобальний простір.

**📚 Технічне пояснення**

В JavaScript, коли ми оголошуємо функції чи змінні в глобальному просторі, вони стають доступними з будь-якої частини коду. Це може призвести до конфліктів, особливо у великих проєктах.

IIFE вирішує цю проблему за допомогою:

1.Створення власного простору (scope): Змінні та функції всередині нього не впливають на глобальний простір.

2.Виконується автоматично: Немає потреби викликати функцію окремо, вона запускається відразу після того, як була визначена.

// Класичний IIFE

(function () {

const secret = "Я є секретною змінною!";

console.log(secret); // "Я є секретною змінною!"

})();

// console.log(secret); // ПОМИЛКА: secret не визначено

**🧠 Чому використовувати IIFE?**

Запобігання забрудненню глобального простору (scope): Внутрішні змінні залишаються захищеними.

Автоматичне виконання коду: Корисно для швидких ініціалізацій.

Приватність змінних: Захищає чутливі дані в межах локального простору (scope).

1. **Як працюють шаблонні рядки (template literals) в JavaScript?**

Template Strings allow both single and double quotes inside a string:

let text = `He's often called "Johnny"`;

Template Strings allow multiline strings:

let text =  
`The quick  
brown fox  
jumps over  
the lazy dog`;

Template String provide an easy way to interpolate variables and expressions into strings.

The method is called string interpolation.

The syntax is: ${...}

Template Strings allow variables in strings:

let firstName = "John";  
let lastName = "Doe";  
  
let text = `Welcome ${firstName}, ${lastName}!`;

**How They Work**

**Parsing:** When the JavaScript engine encounters a template literal, it parses the string into literal parts and expression parts.

**Evaluation:** The expressions inside ${...} are evaluated, and their results are converted to strings and inserted between the literal parts.

**Resulting String:** The engine concatenates everything into a final string.

1. **Що таке деструктуризація (destructuring) об'єктів та масивів в JavaScript?**

The destructuring assignment syntax unpack object properties into variables:let {firstName, lastName} = person;

It can also unpack arrays and any other iterables:

let [firstName, lastName] = person;

1. **Як працюють розширення об'єктів (object spread) та оператори залишку (rest operators)?**Introduction to the JavaScript object spread Operator

In ES6, you use the spread operator (...) to unpack elements of an array. The spread operator can be very useful to clone an array.

**For example:**

let colors = ['red', 'green', 'blue'];

let rgb = [...colors];

console.log(rgb);  
**Output:**

[ 'red', 'green', 'blue' ]   
  
In this example, the spread operator (...) unpacks elements of the colors array and places them in a new array rgb.

**The spread operator (...) can be used to merge two or more arrays** into one as shown in the following example:

let rgb = [ 'red', 'green', 'blue' ];

let cmyk = ['cyan', 'magenta', 'yellow', 'black'];

let merge = [...rgb, ...cmyk];

**Output:**

[ 'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'black' ]

ES2018 expands the spread operator (...) to make it work with its own enumerable properties of an object.  
  
For example, suppose that you have a circle object with one property radius:  
const circle = {

radius: 10

};

The following example uses the spread operator (...) to create an coloredCircle object that has all the properties of the circle object and an additional property color:

const coloredCircle = {

...circle,

color: 'black'

};

console.log(coloredCircle);

**JavaScript Object spread operator use cases**

**1. Clone an object**

const circle = {

radius: 10

};

const clonedCircle = {...circle};

console.log(clonedCircle);

**2.Merging objects**

const circle = {

radius: 10

};

const style = {

backgroundColor: 'red'

};

const solidCircle = {

...circle,

...style

};

console.log(solidCircle);

The last feature you will learn in this article is the rest parameter syntax. The syntax appears the same as spread (...) but has the opposite effect. Instead of unpacking an array or object into individual values, the rest syntax will create an array of an indefinite number of arguments.

In the function restTest for example, if we wanted args to be an array composed of an indefinite number of arguments, we could have the following:

function restTest(...args) {

console.log(args)

}

restTest(1, 2, 3, 4, 5, 6)

**Output**

[1, 2, 3, 4, 5, 6]

Rest syntax can be used as the only parameter or as the last parameter in the list. If used as the only parameter, it will gather all arguments, but if it’s at the end of a list, it will gather every argument that is remaining, as seen in this example:

function restTest(one, two, ...args) {

console.log(one)

console.log(two)

console.log(args)

}

restTest(1, 2, 3, 4, 5, 6)

**Output**

1

2

[3, 4, 5, 6]

1. **Що таке символи (symbols) та як вони використовуються в JavaScript?**