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 в JavaScript?**

**🔍 Як зробити глибоку копію об'єкта в JavaScript?**

**📌 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.

**✅ 1. Глибока копія через structuredClone() (сучасний метод)**

Цей метод створює **повну глибоку копію** об'єкта **без втрати типів даних**.

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).**

**✅ 2. Глибока копія через JSON.parse(JSON.stringify(obj)) (простий спосіб)**

Цей метод спочатку перетворює об'єкт у JSON-рядок, а потім знову в об'єкт.

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.**

**✅ 3. Глибока копія через бібліотеку lodash**

Якщо використовуєте бібліотеку **Lodash**, можна скористатися \_.cloneDeep().

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.**

**✅ 4. Глибока копія через рекурсію (ручний спосіб)**

Якщо потрібно зробити свою реалізацію:

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.**

**🏆 Висновок**

| **Метод** | **Простота** | **Швидкість** | **Підтримка складних типів (Date, Map, Set)** |
| --- | --- | --- | --- |
| structuredClone() | ⭐⭐⭐ | ⭐⭐⭐ | ✅ |
| JSON.parse(JSON.stringify(obj)) | ⭐⭐⭐ | ⭐⭐⭐ | ❌ |
| \_.cloneDeep() (Lodash) | ⭐⭐⭐ | ⭐⭐ | ✅ |
| Рекурсія вручну | ⭐ | ⭐ | ✅ |

✔ **Якщо потрібно підтримувати всі типи даних → structuredClone() або \_.cloneDeep().**  
✔ **Якщо потрібен простий метод → JSON.parse(JSON.stringify(obj)).**  
✔ **Якщо без сторонніх бібліотек → рекурсія.** 🔍 **Що таке замикання (Closures) в JavaScript?**

### 📌 ****Визначення****

**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.

## ✅ ****Як працюють замикання?****

### ****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?**

1. outer() виконується та повертає внутрішню функцію inner.
2. inner() "запам’ятовує" змінну message, навіть якщо outer() вже завершила виконання.
3. myClosure() викликає inner(), і вона все ще має доступ до message.

## ✅ ****Замикання для збереження стану****

### ****Приклад 2: Лічильник****

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)****

### ****Приклад 3: Інкапсуляція****

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.

## ✅ ****Замикання всередині**** for

### ****Помилка при використанні**** var

for (var i = 1; i <= 3; i++) {

setTimeout(() => console.log(i), 1000);

}

// Виведе: 4, 4, 4

❌ **Чому так?**

* var має **функціональну область видимості**, а не блочну.
* setTimeout() виконується **після завершення циклу**, коли i = 4.

### ****Правильний спосіб (з**** let****)****

for (let i = 1; i <= 3; i++) {

setTimeout(() => console.log(i), 1000);

}

// Виведе: 1, 2, 3

✔ **Чому працює?**

* let має **блочну область видимості**, тому кожна ітерація створює **нове замикання**.

## 🔥 ****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

## 🎯 ****Висновок****

1️ **Замикання дозволяють функціям "запам’ятовувати" зовнішні змінні.**  
2️ **Вони корисні для створення приватних змінних та інкапсуляції даних.**  
3️ **Застосовуються в лічильниках, обробниках подій, модулях та циклах.**

ℹ️ **Головне правило:** якщо функція має доступ до змінної після її області видимості, це **замикання**! 🚀