Розширена робота з функціями в JavaScript

Рекурсія, стек, rest/spread, область видимості й замикання, var, globalThis, об’єкт функції/NFE, new Function, таймери, декоратори, call/apply/bind, стрілкові функції

План лекції

1. Рекурсія та стек
2. Залишкові параметри та синтаксис поширення
3. Область видимості змінної, замикання
4. Застаріле ключове слово «var»
5. Глобальний об’єкт
6. Об’єкт функції, NFE
7. Синтаксис «new Function»
8. Планування: setTimeout та setInterval
9. Декоратори та переадресація виклику, call/apply
10. Прив’язка контексту до функції
11. Повторення стрілкових функцій

П.1. Рекурсія та стек в JavaScript

План по темі «Рекурсія та стек»

- Що таке рекурсія: база та крок
- Два підходи: ітеративний vs рекурсивний (pow)
- Контекст виконання та стек викликів
- Рекурсивний обхід складних структур (приклад «сума зарплат»)
- Рекурсивні структури даних: дерева та зв’язаний список
- Коли обирати рекурсію: переваги/недоліки

Рекурсія: визначення

Рекурсія — це прийом, коли функція викликає саму себе для розв’язання задачі шляхом зведення її до простіших підзадач.

Ключові поняття: база рекурсії (миттєво повертає відповідь) і рекурсивний крок (спрощує задачу та викликає функцію ще раз).

Типові сфери застосування: робота з деревами/графами, розбір вкладених структур, пошук, обходи, генератори комбінацій.

Приклад: піднесення до степеня (ітеративно)

```
function pow(x, n) {
  let result = 1;
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  return result;
}
```

Один контекст виконання; пам’ять не залежить від n.
Підходить, коли прямий цикл простіший за рекурсію.

Приклад: піднесення до степеня (рекурсивно)

```
function pow(x, n) {
  if (n == 1) {
    return x;                       // база
  } else {
    return x * pow(x, n - 1);       // рекурсивний крок
  }
}
```

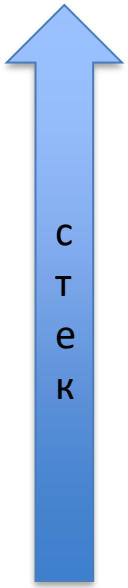
Скорочує задачу xn → x·x(n-1) до бази n==1.
Код коротший, але створює глибину рекурсії n.

Схема розгалуження рекурсії для pow(x, n)

pow(x, n): якщо n==1 → x

pow(x, n): інакше → x * pow(x, n-1)

…повторювати, поки n зменшиться до 1



Контекст виконання та стек

Під час кожного виклику створюється контекст виконання з локальними змінними, параметрами та позицією виконання.

При вкладеному виклику поточний контекст кладе в стек, далі створюється новий; після завершення — попередній контекст відновлюється.

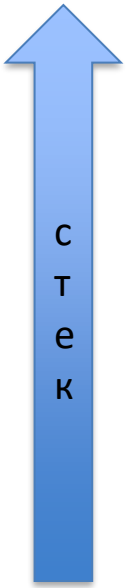
Глибина рекурсії дорівнює максимальній кількості контекстів у стеці під час виконання.

Кроки pow(2, 3): стан стека

Контекст: {x:2, n:3} → рядок if/else

Контекст: {x:2, n:2} → підвиклик

Контекст: {x:2, n:1} → база → повертає 2



Розмотування стека (unwinding)

Після повернення з бази виконання відновлюється у попередніх контекстах:
pow(2,1) → 2; підставляємо у pow(2,2) → 2*2=4; у pow(2,3) → 2*4=8.

Ітеративний варіант споживає сталу пам’ять; рекурсивний — O(n) контекстів.

Обмеження та зауваги

- Максимальна глибина рекурсії залежить від рушія (безпечний порядок ~10k викликів).
- Для дуже глибоких задач — перепишіть на цикл або застосуйте техніки, що зменшують глибину.
- Рекурсія зручна для вкладених структур; цикл — для лінійних задач.

Рекурсивний обхід: ідея

Якщо структура складається з частин того ж типу (дерево підрозділів, DOM), природно визначити базовий випадок (лист) і рекурсивний крок (вузол з підвузлами).

Приклад: обчислити суму зарплат у компанії, де кожен підрозділ — або масив співробітників (лист), або об’єкт з підвідділами (вузол).

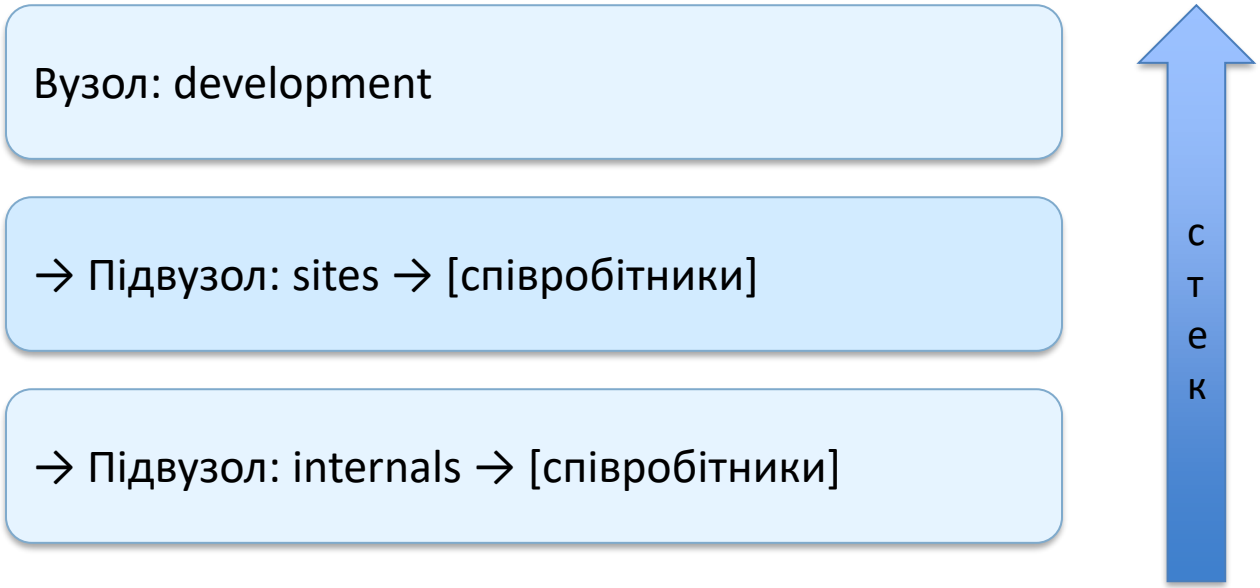
Приклад: сума зарплат (рекурсивний обхід)

```
function sumSalaries(department) {
  if (Array.isArray(department)) {           // лист
    return department.reduce((s, p) => s + p.salary, 0);
  } else {                                   // вузол
    let sum = 0;
    for (let sub of Object.values(department)) {
      sum += sumSalaries(sub);
    }
    return sum;
  }
}
```

Працює для довільної глибини вкладення.

База: масив людей; рекурсивний крок: обхід підвідділів.

Дерево викликів для суми зарплат



Рекурсивні структури даних

Структура рекурсивна, якщо в описі використовує сама себе:

• Відділ компанії — масив людей або об’єкт із підвідділами.
• DOM/HTML — тег містить інші теги/вузли тексту/коментарі.
• Зв’язаний список — елемент з посиланням next на такий самий елемент або null.

Зв’язаний список: приклад структури

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};
```

Вставка/видалення в середині — O(1) (за наявності посилання).
Доступ за індексом — O(n), на відміну від масиву.

Операції над списком (вставка/видалення)

```
// додати на початок
list = { value: "new", next: list };

// видалити другий елемент
list.next = list.next.next;
```

Маніпулюємо посиланнями next без «масових» зрушень, як у масивах.

Коли обирати рекурсію

- Коли структура природно дерево-/вкладена.
- Коли рекурсивний код суттєво коротший і зрозуміліший.
- Коли глибина обмежена й безпечна для середовища виконання.

Типові підводні камені

- Надмірна глибина → помилка переповнення стека.
- Повторне обчислення підзадач (допомагають кешування/мемоізація).
- Неправильно визначена база рекурсії → нескінченна рекурсія.

Практичні завдання для самостійного виконання

1) Реалізуйте pow(x, n) двома способами та порівняйте швидкодію.

2) Напишіть sumSalaries(company) та додайте нові рівні вкладення.

3) Створіть зв’язаний список і реалізуйте print/listReverse рекурсивно й ітеративно.

4) Дослідіть межі глибини рекурсії у вашому оточенні.

П.2. Залишкові параметри та синтаксис розширення (Rest & Spread)

План до теми «Залишкові параметри та синтаксис розширення»

- Залишкові параметри: ...rest
- Псевдомасив arguments
- Синтаксис розширення: ...spread
- Array.from vs spread
- Копіювання масивів/об’єктів

Ідея залишкових параметрів

- Функція може приймати довільну кількість аргументів.
- `...rest` збирає «зайві» аргументи у масив.
- Позиція важлива: `...rest` завжди останній параметр.
- Типові задачі: сумування, агрегації, прокидання параметрів.

Псевдомасив arguments

- Доступний у звичайних функціях, містить усі передані аргументи.
- Є ітерованим та псевдомасивом, але НЕ справжнім масивом.
- Не підтримує методи масиву напряму (map, filter тощо).
- Стрілкові функції не мають власного arguments – беруть зовнішній.

Синтаксис розширення (spread)

- Розгортає ітероване у список окремих значень.
- Працює в контексті виклику функції, літералів масивів/об’єктів.
- Зручно для Math.max(...arr), конкатенації масивів, клонування.

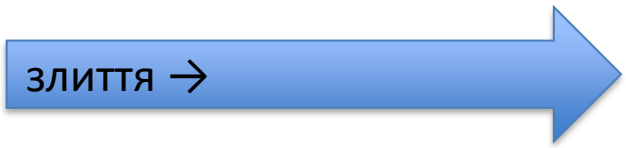
Схема: "..." у ролях rest vs spread

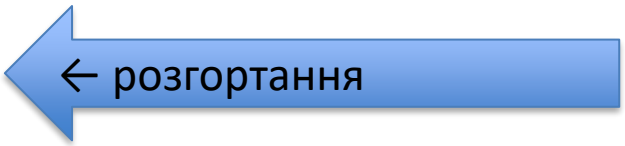
```
function fn(a, ...rest) { /* rest: масив */ }

fn(...arr) // spread: список аргументів
```

...rest → збирає аргументи у масив

...spread → розгортає масив у список





Практика для самостійного виконання

- Напишіть sumAll(...nums), що обчислює суму.
- Реалізуйте maxOfArrays(...arrays) → Math.max по всіх елементах.
- Створіть поверхневе клонування userCopy = {...user}.
- Перетворіть псевдомасив arguments у справжній масив за допомогою Array.from.

Висновки до п.2.

- `...rest` у сигнатурі → отримуємо масив додаткових аргументів.
- `...spread` у викликах/літералах → розгортає ітеровані структури.
- Array.from універсальніший за spread для псевдомасивів.
- Клонування через spread — поверхневе.

П.3. Область видимості змінної та замикання (Closures) у JavaScript

План до теми «Область видимості змінної та замикання»

- Блокова область видимості: let/const
- Вкладені функції та повернення функції
- Лексичне середовище: запис середовища й посилання назовні
- Замикання: визначення та інтуїція
- Практичні приклади: лічильники, фабрики функцій
- Пам’ять, збирач сміття та оптимізації рушіїв

Блоки коду: область видимості let/const

- Змінні, оголошені в блоці {…}, доступні тільки всередині блоку.
- Аналогічно працюють гілки if/else та тіла циклів.

```
{
  let msg = "Привіт";
  console.log(msg);
}
console.log(typeof msg); // ReferenceError у реальному коді
```

Приклади: if / for створюють власну область

- Конструкція for створює власну область для лічильника.
- Код поза блоком не бачить внутрішні змінні.

```
if (true) {
  let phrase = "Hello!";
  console.log(phrase); // видно
}
for (let i = 0; i < 3; i++) {
  // i видно лише тут
}
console.log(typeof i); // 'undefined'
```

Вкладені функції

- Вкладена функція має доступ до змінних зовнішньої.
- Корисно виділяти допоміжні обчислення.

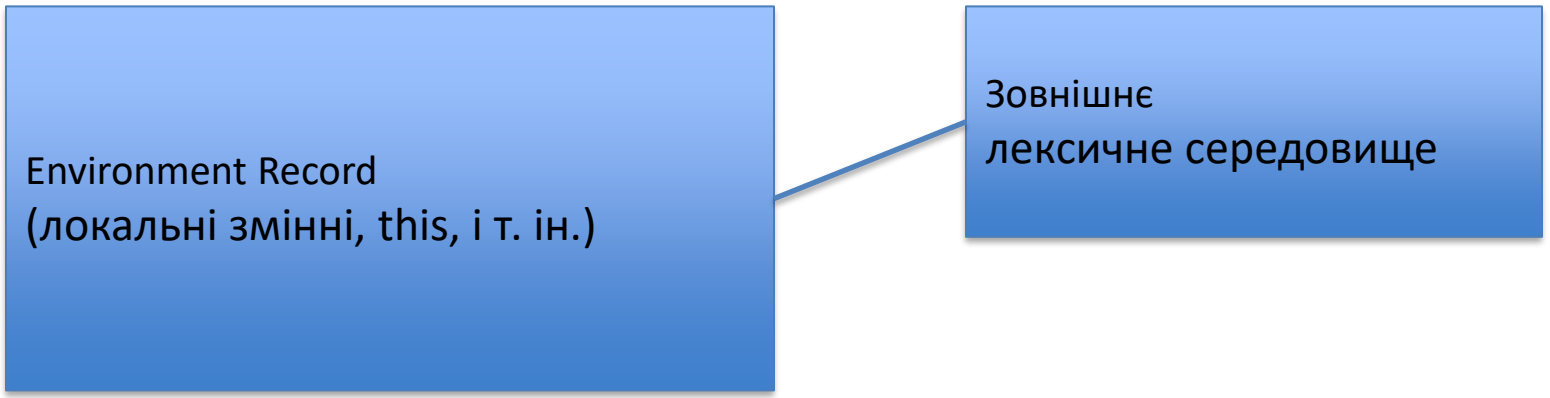
```
function sayHiBye(first, last) {
  function full() { return first + " " + last; }
  console.log("Привіт, " + full());
  console.log("Бувай, " + full());
}
```

Повернення функції (фабрика)

- Повернена функція «пам’ятає» середовище створення.
- Кожен виклик makeCounter створює власний лічильник.

```
function makeCounter() {
  let count = 0;
  return function () {
    return count++; // доступ до зовнішньої змінної
  };
}
const c1 = makeCounter();
console.log(c1()); // 0
console.log(c1()); // 1
```

Лексичне середовище: структура



Function Declaration і ініціалізація

- Оголошені як Function Declaration — готові до виклику одразу.
- Function Expression ініціалізуються під час виконання виразу.
- Під час старту виконання середовище попередньо знає про оголошення.

Внутрішнє/зовнішнє середовище при виклику

- На кожен виклик створюється новий запис (локальні змінні, параметри).
- Пошук іде ланцюжком: спочатку локальні → далі назовні → глобальне.
- Якщо змінну ніде не знайдено: у strict-mode — помилка.

Замикання: визначення та властивості

- Функція, що запам’ятовує зовнішні змінні місця створення.
- Доступ зберігається навіть після завершення зовнішньої функції.
- Технічно: прихована властивість [[Environment]] з посиланням на середовище.

Замикання на прикладі лічильника

- Кожен екземпляр має власний стан n.
- Стан інкапсульований та недоступний напряму.

```
function makeCounter() {
  let n = 0;
  return () => ++n;
}
const a = makeCounter();
const b = makeCounter();
console.log(a()); // 1
console.log(a()); // 2
console.log(b()); // 1  (незалежний стан)
```

Незалежні лічильники

- Кожен виклик фабрики створює новий ланцюжок лексичних середовищ.
- Стан одного лічильника не впливає на інший.

Життєвий цикл і збирач сміття

- Лексичне середовище «живе», доки на нього є посилання (через функцію).
- Після втрати посилань — середовище доступне для GC.

```
function f() {
  let value = 123;
  return function() { console.log(value); };
}
let g = f(); // середовище зберігається, поки доступна g
g = null;    // тепер середовище може бути зібране GC
```

Оптимізації рушіїв і налагодження

- Рушій може викидати неиспользувані змінні із замикання для економії пам’яті.
- У DevTools це може виглядати як «змінної немає» всередині debugger.
- Це очікувана оптимізація (приклад у V8/Chrome).

Задача: що виведе функція?

- Питання на розуміння моменту читання змінної під час виклику.

```
let name = "Іван";
function sayHi() { console.log("Привіт, " + name);
}
name = "Петро";
sayHi(); // ?
```

Задача: місце створення проти місця виклику

- Які змінні будуть доступні? (Місце створення визначає середовище)

```
function makeWorker() {
  let name = "Петро";
  return function() { console.log(name); };
}
let name = "Іван";
let work = makeWorker();
work(); // ?
```

Задача: незалежні лічильники

- Перевірка незалежності внутрішніх станів.

```
function makeCounter() {
  let count = 0;
  return function() { return count++; };
}
const c1 = makeCounter();
const c2 = makeCounter();
console.log(c1(), c1()); // ?
console.log(c2(), c2()); // ?
```

Задача: up/down у конструкторі

- Чи спільний стан використовують методи up/down?

```
function Counter() {
  let count = 0;
  this.up   = () => ++count;
  this.down = () => --count;
}
const counter = new Counter();
console.log(counter.up());   // ?
console.log(counter.up());   // ?
console.log(counter.down()); // ?
```

Рекомендації щодо використання замикань

- Тримайте замкнений стан мінімальним і очевидним.
- Уникайте витоків пам’яті: скасовуйте таймери/посилання, коли більше не потрібні.
- Для налагодження використовуйте breakpoints і уважно дивіться на область видимості в DevTools.

П.4. JavaScript: ключове слово var (застаріле)

План до теми «ключове слово var »

- Що таке var: історичний контекст
- Відсутність блочної області видимості
- var у циклах та в функціях
- Підняття (hoisting): оголошення vs присвоєння
- Повторні оголошення (re-declaration)
- IIFE: навіщо використовували раніше
- Порівняння з let/const та практичні поради

Що таке var

- Старий спосіб оголошення змінних у JS.
- Сьогодні у сучасному коді майже не використовується.
- Зустрічається в старих проєктах – важливо розуміти відмінності.

Відсутність блочної області видимості

Змінні, оголошені через var, «бачаться» за межами блоку if/for/while. Це головна відмінність від let/const.

```
if (true) {
  var test = true; // var ігнорує блок
}

alert(test); // true – змінна існує поза блоком

// з let:
// if (true) {
//   let test = true;
// }
// alert(test); // ReferenceError
```

var у циклах

var в for створює змінні рівня функції/глобальні, тому після циклу вони доступні.

```
for (var i = 0; i < 3; i++) {
  var one = 1;
}

alert(i);   // 3 – видно поза циклом
alert(one); // 1 – також видно поза циклом
```

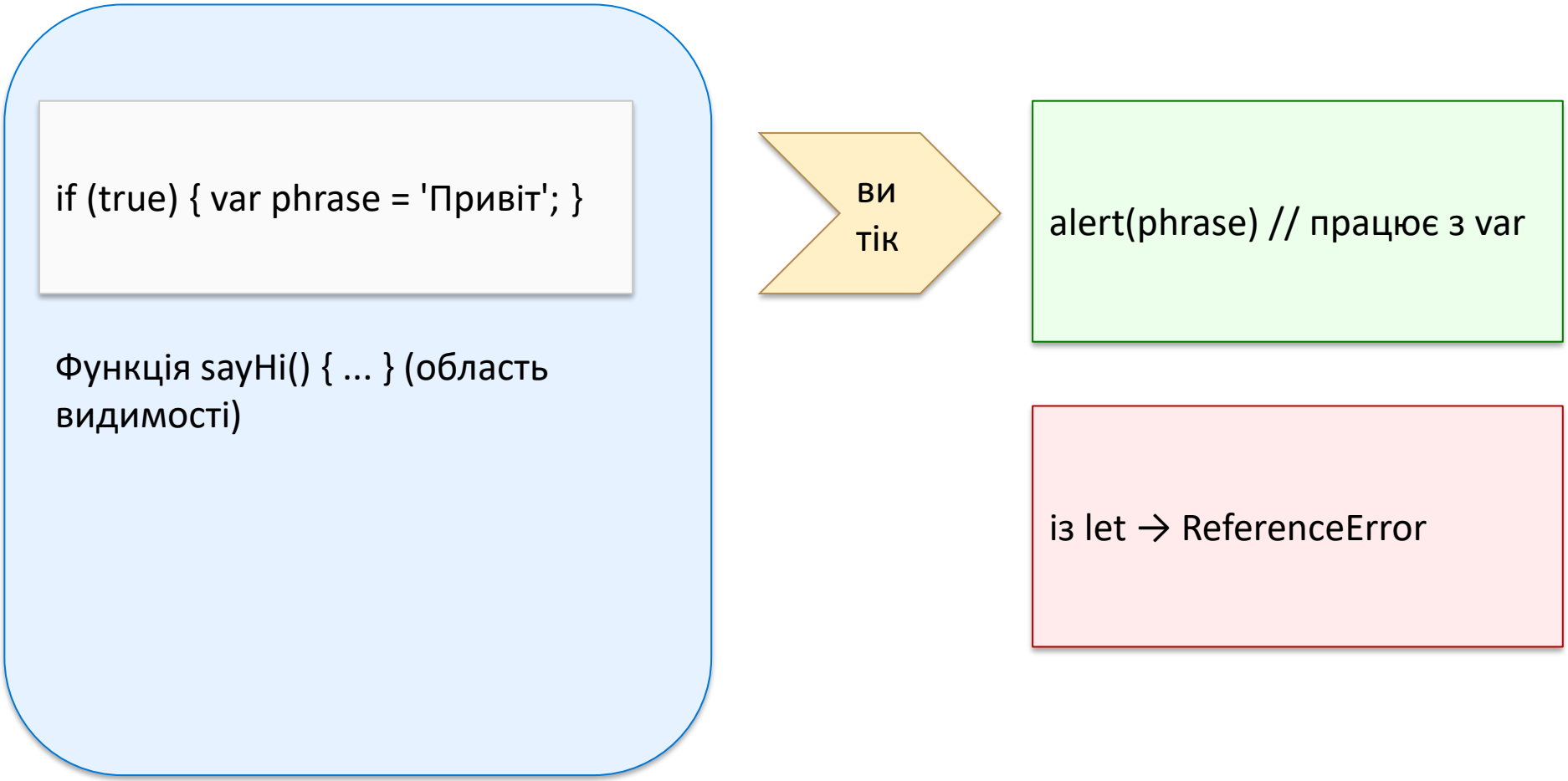
var у функціях: функціональна область

Всередині функції var має область видимості функції. За межами функції недоступна.

```
function sayHi() {
  if (true) {
    var phrase = 'Привіт';
  }
  alert(phrase); // OK
}

sayHi();
alert(phrase); // ReferenceError поза функцією
```

Схема: блок проти функції (var vs let)



Підняття (hoisting): ключові моменти

- Оголошення var обробляються на старті виконання функції/скрипту.
- Оголошення піднімаються, присвоєння – ні.
- До присвоєння змінна має значення undefined (але вже існує).

Приклад hoisting (1)

Оголошення піднімається, тому звернення до змінної можливе, але значення – undefined до присвоєння.

```
function sayHi() {
  alert(phrase); // undefined
  var phrase = 'Привіт';
}

sayHi();
```

Схема hoisting: оголошення vs присвоєння

Джерело:

```
function sayHi() {
  alert(phrase);
  var phrase = 'Привіт';
}
```

Після підняття (hoisting):

```
function sayHi() {
  var phrase; // оголошення спочатку
  alert(phrase); // undefined
  phrase = 'Привіт'; // присвоєння
пізніше
}
```

Приклад hoisting (2): if(false) не рятує

Навіть якщо гілка коду ніколи не виконається, оголошення var усередині неї буде «підняте».

```
function sayHi() {
  phrase = 'Привіт'; // (*)
  if (false) {
    var phrase;
  }
  alert(phrase); // 'Привіт'
}

sayHi();
```

Повторні оголошення (re-declaration)

На відміну від let/const, var допускає повторне оголошення змінної в одній області без помилки.

```
var user = 'Петро';
var user = 'Іван'; // друге оголошення не кидає помилку
alert(user); // 'Іван'
```

Оголошення після використання

Через hoisting ми можемо звертатись до var до рядка з оголошенням (значення буде undefined до присвоєння).

```
function demo() {
  alert(x);   // undefined
  var x = 42; // оголошення піднято, присвоєння — ні
}

demo();
```

IIFE (Immediately-Invoked Function Expression)

Раніше IIFE використовували, щоб створити «приватну» область видимості, обходячи відсутність блочної області в var.

```
(function() {
  var message = 'Привіт';
  alert(message);
})();

// Варіанти синтаксису:
// (function(){})()
// !function(){}()
// +function(){}()
```

Чому сьогодні НЕ варто використовувати var

- Відсутність блочної області → легше припуститись витоків змінних.
- Поведінка hoisting ускладнює читання і налагодження.
- let/const надійніші та передбачуваніші для сучасного JS.

Практичні поради

- Використовуйте const за замовчуванням; let – коли потрібно перевизначення.
- var залишайте для підтримки старих скриптів.
- Під час міграції рахуйтеся з hoisting та повторними оголошеннями.

П. 5. Глобальний об’єкт у JavaScript

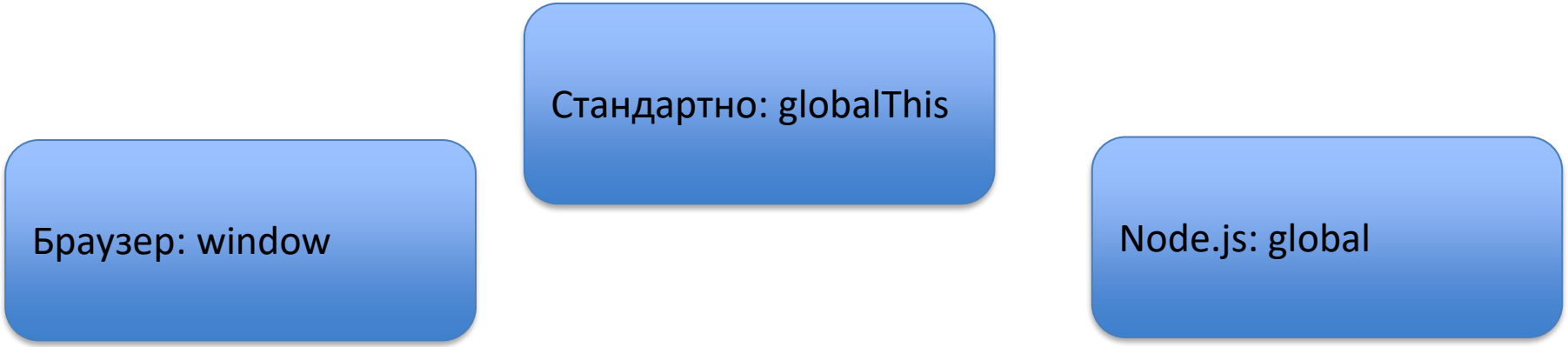
План до теми «Глобальний об’єкт»

- Що таке глобальний об’єкт
- Назви у різних середовищах: window / global / globalThis
- Доступ напряму та через префікс window
- var/function vs let/const: що стає властивістю глобального об’єкта
- Усвідомлені глобальні значення
- Перевірка підтримки та поліфіли
- Поради та антипатерни

Що таке глобальний об’єкт

- Центральний об’єкт середовища виконання, чиї властивості доступні скрізь у програмі.
- У браузері містить вбудовані значення (наприклад, Array) та дані середовища (window.innerHeight).
- У Node.js свій глобальний об’єкт середовища.
- Універсальна стандартизована назва — globalThis.

Назви глобального об’єкта у різних середовищах



globalThis — уніфікований спосіб доступу до глобального об’єкта в різних середовищах.

Доступ: напряму та через window

```
alert("Привіт");
// те саме, що й
window.alert("Привіт");
```

У браузері методи/властивості глобального об’єкта зазвичай доступні без префікса.

var/function vs let/const

```
var gVar = 5;
function f() {}

console.log(window.gVar); // 5
console.log(typeof window.f); // "function"

let gLet = 5;
console.log(window.gLet); // undefined
```

У браузері var та оголошення функцій стають властивостями window; let/const — ні.

Усвідомлені глобальні значення

```
window.currentUser = { name: "John" };

// далі у коді
console.log(currentUser.name);      // John
console.log(window.currentUser.name); // John
```

Зберігайте глобальні значення лише коли це справді потрібно; мінімізуйте глобальний стан.

Перевірка підтримки та поліфіли

```
if (!window.Promise) {
  console.warn("Твій браузер застарів!");
  // Можна підключити поліфіл або надати спрощену реалізацію
  // window.Promise = ...
}

// Кросс-середовищно краще використовувати globalThis:
if (!globalThis.Promise) { /* поліфіл */ }
```

Користуйтеся перевіркою наявності властивостей; за потреби — поліфіли.

Поради та антипатерни

- Уникайте «забруднення» глобального простору імен; використовуйте модулі.
- Не покладайтеся на те, що var/Function Declaration додають властивості до window.
- Використовуйте globalThis для кросплатформеного доступу до глобального об’єкта.
- Звертайтеся до глобальних властивостей явно (window.x / globalThis.x), коли це справді потрібно.

П.6. Об’єкт функції та Named Function Expression (NFE)

Функція як об’єкт у JavaScript

- Функції — це об’єкти, які можна викликати: мають власні властивості та методи.
- До функції можна додавати кастомні властивості (§ name, length — вбудовані).
- Функцію зручно використовувати як «простір імен»: додавати допоміжні методи.
- Бібліотеки (jQuery $, lodash _) — функції з набором утиліт як властивості.

```
function sayHi() { alert("Привіт"); }
sayHi.custom = 123;           // власна властивість
console.log(typeof sayHi);    // "function"
console.log(sayHi.custom);    // 123
```

Властивість функції name

- name — ім’я функції (береться з оголошення або визначається з контексту).
- Методи об’єктів також мають коректне name за замовчуванням.
- Корисно для діагностики, логування та відлагодження.

```
function f() {}
console.log(f.name); // "f"

let say = function() {};
console.log(say.name); // "say" (визначено з контексту присвоєння)

const obj = {
  greet() {},
  bye: function() {}
};
console.log(obj.greet.name); // "greet"
console.log(obj.bye.name);   // "bye"
```

Властивість length (кількість оголошених параметрів)

- length — число формальних параметрів функції (rest-параметри не рахуються).
- Може застосовуватись для простого поліморфізму за сигнатурою.

```
function f1(a) {}
function f2(a, b) {}
function many(a, b, ...rest) {}
console.log(f1.length, f2.length, many.length); // 1 2 2

function ask(question, ...handlers) {
  const ok = confirm(question);
  for (const h of handlers) {
    if (h.length === 0) { if (ok) h(); } else { h(ok); }
  }
}
```

Кастомні властивості як альтернатива замиканням

- Ідея: зберігати стан прямо на функції як на об’єкті (наприклад, counter.count).
- Плюс: доступний/видимий стан; Мінус: зовнішній код може змінити значення.
- Замикання приховує стан усередині лексичного середовища — без прямого доступу.

```
function makeCounter() {
  function counter() { return counter.count++; }
  counter.count = 0;
  return counter;
}
const c = makeCounter();
console.log(c()); // 0
console.log(c()); // 1
c.count = 10;     // зовнішній код може змінити стан
console.log(c()); // 10
```

Named Function Expression (NFE): що це таке

- NFE — Function Expression з внутрішнім ім’ям (лише всередині самої функції).
- Дозволяє надійно посилатися на себе (рекурсія, повторний виклик).
- Внутрішнє ім’я недоступне ззовні та не засмічує простір імен.

```
let sayHi = function func(who) {
  if (who) { alert(`Привіт, ${who}`); }
  else { func("Гість"); } // самопосилання на внутрішнє ім’я
};
sayHi(); // "Привіт, Гість"
// func(); // Помилка: недоступна ззовні
```

Навіщо NFE, якщо можна sayHi() всередині?

- Якщо зовнішня змінна зміниться (переприсвоєння), внутрішній виклик зламається.
- NFE гарантує стабільне самопосилання незалежно від зовнішніх змін.

```
let sayHi = function(who) {
  if (!who) sayHi("Гість"); // залежить від зовнішньої змінної
};
let welcome = sayHi;
sayHi = null;
// welcome(); // Падіння: sayHi більше не функція

// З NFE працює стабільно (див. попередній слайд).
```

Патерн: функція + допоміжні методи у властивостях

- Головна функція виконує основну дію; допоміжні утиліти підвішені як властивості.
- Зменшує кількість глобальних імен, зручно організовувати API.

```
function _() { /* ядро */ }
_.keyBy = function(arr, key) { /* ... */ };
_.clone = function(obj) { /* ... */ };
// Виклик:
_();
_.keyBy([{id: 1}], "id");
```

Коли обирати замикання, властивості або NFE

- Стан має бути приватним → замикання.
- Стан має бути видимим/конфігурованим → властивість функції.
- Потрібне надійне самопосилання всередині виразу → NFE.

```
// Резюме вибору інструмента:
// приватний стан -> замикання
// відкритий стан -> властивість функції
// стабільне самопосилання -> NFE
```

Підсумки та джерела

- Функції — повноцінні об’єкти з властивостями name і length.
- Можна додавати власні властивості для збереження стану або API.
- NFE надає внутрішнє ім’я для безпечного самопосилання.
- Джерело: uk.javascript.info/function-object

П.7. Синтаксис «new Function»

Що таке new Function

- Альтернативний спосіб створення функції з рядка під час виконання.
- Корисно для динамічної компіляції коду (шаблони, отриманий із сервера код).
- На відміну від оголошень/виразів функцій, тіло передається як текст.
- Використовується рідко, але інколи не має прямої альтернативи.

Синтаксис і базовий приклад

- Синтаксис: new Function([arg1, arg2, …argN], functionBody).
- Аргументи можна задавати списком або одним рядком через кому.
- Функція повертає викликуваний об’єкт- функцію.

```
let sum = new Function('a', 'b', 'return a + b');
alert( sum(1, 2) ); // 3
```

Приклад без аргументів

- Можна створити функцію лише з тілом без параметрів.
- Тіло функції – звичайний JavaScript-код у рядку.

```
let sayHi = new Function('alert("Привіт")');
sayHi(); // Привіт
```

Коли доречно застосовувати

- Код надходить під час виконання (динамічні плагіни/шаблони).
- Післядеплойні сценарії: завантаження обчислюваних формул/правил.
- Будьте обережні з безпекою: виконує довірений код лише з перевірених джерел.

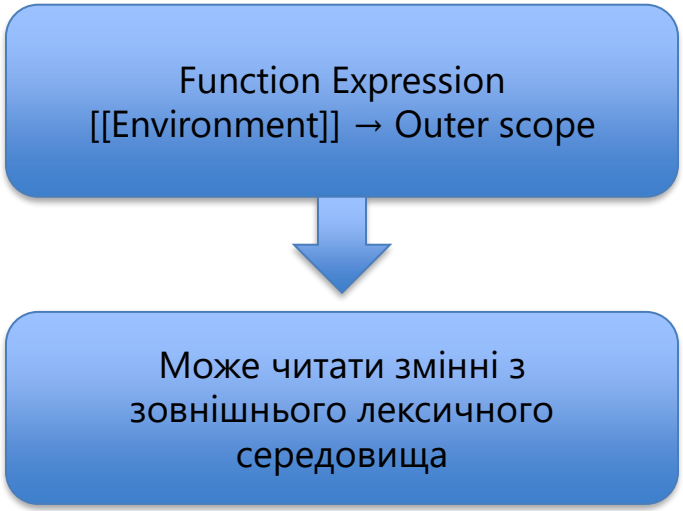
Особливість замикань ([[Environment]])

- Функції, створені через new Function, мають [[Environment]] = глобальна область.
- Вони НЕ бачать змінних із зовнішніх лексичних середовищ.
- Доступні лише глобальні змінні та передані аргументи.

```
function getFunc() {
  let value = 'тест';
  let func = new Function('alert(value)');
  return func;
}
getFunc()(); // Помилка: value не визначено
```

Порівняння: звичайна функція бачить зовнішні змінні

- Function Expression/Declaration: [[Environment]] → зовнішня область.
- Може читати value із охоплювального лексичного середовища.




```
function getFunc() {
  let value = 'тест';
  let func = function() { alert(value); };
  return func;
}
getFunc()(); // «тест»
```

Чому це зроблено саме так

- Мінімізатори змінюють локальні назви змінних; доступ «назовні» з динамічного коду ненадійний.
- Явна передача через аргументи прозоріша й безпечніша.
- Зменшує зв’язність між динамічним фрагментом і основним кодом.

Практичні нотатки

- Передавайте дані через аргументи: new Function("a,b", "return a+b").
- Стежте за безпекою: не виконуйте неперевірений код користувача.
- Використовуйте лінтінг/типізацію для статичного коду; динамічні частини – мінімальні.

```
let buildSum = new Function('a,b', 'return a + b');
console.log(buildSum(2, 5)); // 7
```

Підсумки

- new Function створює функцію з рядка під час виконання.
- Її [[Environment]] – глобальний, зовнішніх змінних не видно.
- Основні кейси: динамічні шаблони/плагіни/формули.
- Дані передавайте аргументами; дотримуйтесь безпеки.
- Джерело: uk.javascript.info/new-function

Самостійно освоїти:

П.8. Планування: setTimeout та setInterval
П.9. Декоратори та переадресація виклику, call/apply
П.10. Прив’язка контексту до функції
П.11. Повторення стрілкових функцій

Дякую за увагу!