**Повернення функції із функцій & IIFE**

У JavaScript ми можемо створити функцію, яка повертає іншу функцію. Це дає можливість динамічно створювати та використовувати функції в залежності від потреб програми. Повернення функції із функцій використовується для реалізації різноманітних патернів програмування, зокрема для створення фабрик функцій, конфігурування функцій із певними параметрами та іншого.

Приклад:

function createGreeting(greeting) {

return function(name) {

return greeting + ', ' + name + '!';

};

}

var greetHello = createGreeting('Hello');

var greetHi = createGreeting('Hi');

console.log(greetHello('Alice')); // Виведе: "Hello, Alice!"

console.log(greetHi('Bob')); // Виведе: "Hi, Bob!"

У цьому прикладі, функція createGreeting приймає рядок greeting і повертає нову функцію, яка, в свою чергу, приймає ім'я name і повертає привітання з цим ім'ям. Це дозволяє нам створювати різні варіанти привітань заздалегідь налаштовані з певним текстом привітання, які потім можна використовувати для генерації персоналізованих привітань.

Такий підхід дозволяє нам зробити код більш гнучким і модульним, оскільки ми можемо легко створювати та використовувати функції, налаштовані для конкретних завдань, без необхідності дублювати код.

### Immediately Invoked Function Expression (IIFE)

Immediately Invoked Function Expression (IIFE) — це функція в JavaScript, яка виконується одразу після того, як її оголошено. Ми використовуємо IIFE для створення нової області видимості, що дозволяє нам "сховати" змінні та функції від глобальної області видимості, тим самим запобігаючи конфліктам імен і забезпечуючи кращу структуру коду за принципом модульності.

**Навіщо використовується?**

* **Ізоляція змінних:** IIFE допомагає уникнути забруднення глобальної області видимості, оскільки всі змінні та функції, оголошені всередині IIFE, доступні лише в її межах.
* **Створення приватних змінних та функцій:** за допомогою IIFE можна створити "приватні" змінні та функції, які не можуть бути доступними ззовні.
* **Негайне виконання логіки:** IIFE дозволяє одразу виконати певний блок коду без необхідності його явного виклику.

Приклад коду:

(function() {

var privateVar = "Я приватна змінна";

console.log(privateVar); // Виведе: "Я приватна змінна"

})();

// Спроба доступу до privateVar ззовні IIFE видасть помилку:

console.log(privateVar); // ReferenceError: privateVar is not defined

У цьому прикладі, змінна privateVar оголошена всередині IIFE, тому вона доступна лише всередині цієї функції. Після завершення виконання IIFE, спроба доступу до privateVar ззовні видасть помилку, оскільки змінна не існує в глобальній області видимості.

Використання IIFE є потужним інструментом для створення чистого та організованого коду в JavaScript, особливо коли йдеться про розробку великих або складних програм.

## HoF & Currying & Basic Scope

### Higher-Order Functions (HoF)

Higher-Order Functions (функції вищого порядку) у JavaScript — це функції, які можуть приймати інші функції як аргументи або повертати їх як результат своєї роботи. Ми використовуємо такі функції для створення абстракцій, які дозволяють маніпулювати іншими функціями та їх викликами, що робить наш код більш модульним, гнучким і виразним.

Приклад коду:

**Функція, яка приймає іншу функцію як аргумент:**

function repeat(n, action) {

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

action(i);

}

}

repeat(3, console.log); // Виводить 0, потім 1, потім 2

У цьому прикладі, repeat є функцією вищого порядку, оскільки вона приймає іншу функцію (console.log) як аргумент і викликає її в циклі.

**Функція, яка повертає іншу функцію:**

function greaterThan(n) {

return function(m) {

return m > n;

};

}

var greaterThan10 = greaterThan(10);

console.log(greaterThan10(11)); // Виводить true

Тут greaterThan є функцією вищого порядку, оскільки вона повертає іншу функцію, яка перевіряє, чи є дане число більшим за n.

**Функції вищого порядку дозволяють нам:**

* Створювати абстракції, які можуть маніпулювати іншими діями, зменшуючи повторення коду.
* Легко реалізовувати загальні програмні шаблони, такі як ітерація по елементах, фільтрація даних, перетворення даних тощо.
* Створювати функції, налаштовані на конкретні потреби за допомогою часткового застосування або каррінгу.

Використання функцій вищого порядку є ключовим аспектом функціонального програмування в JavaScript, що сприяє написанню більш чистого, ефективного та легко читабельного коду.

### Currying (Каррування, Каррінг)

Currying — це процес перетворення функції з декількома аргументами в послідовність вкладених функцій, кожна з яких приймає один аргумент. Ми використовуємо currying для створення нової версії функції, яка може бути викликана з частковим набором аргументів, і повертає функцію, яка приймає решту аргументів. Це дозволяє нам створювати більш модульний та перевикористовуваний код.

Приклад коду:

**Функція без currying:**

function add(a, b) {

return a + b;

}

console.log(add(1, 2)); // Виведе 3

Та ж функція з використанням currying:

function add(a) {

return function(b) {

return a + b;

};

}

var addToOne = add(1);

console.log(addToOne(2)); // Виведе 3

У цьому прикладі, за допомогою currying ми перетворили функцію add, яка приймає два аргументи, на функцію, що приймає перший аргумент a і повертає іншу функцію, яка чекає на другий аргумент b. Таким чином, addToOne стає функцією, яка додає 1 до переданого їй числа.

**Currying дозволяє нам:**

* Створювати нові функції з частково застосованими аргументами на основі існуючих функцій, що сприяє перевикористанню коду.
* Розбивати складні функції на простіші функції, кожна з яких виконує лише частину завдання.
* Використовувати функціональний підхід для розв'язання задач, що робить код більш виразним та легким для розуміння.

Currying є важливою концепцією в функціональному програмуванні, яка надає більшу гнучкість при створенні та використанні функцій.

### Basic Scope (області видимості змінних з var)

Поки розглядаємо спрощено, лише для var! let та const будуть пояснені далі!

Scope у програмуванні визначає контекст виконання коду, який окреслює доступність змінних. У JavaScript існують два основних типи областей видимості: глобальна та локальна.

### Глобальна область видимості

Змінні, оголошені поза будь-якими функціями або блоками коду, мають глобальну область видимості. Це означає, що вони доступні з будь-якого місця у коді.

**Приклад:**

var globalVar = "Я глобальна змінна";

function testScope() {

console.log(globalVar); // Доступ до глобальної змінної зсередини функції

}

testScope(); // Виведе: "Я глобальна змінна"

console.log(globalVar); // Доступ до глобальної змінної ззовні функції

### Локальна область видимості

Змінні, оголошені всередині функції, належать до локальної області видимості цієї функції. Це означає, що вони доступні лише всередині цієї функції та не можуть бути доступні ззовні.

**Приклад:**

function testLocalScope() {

var localVar = "Я локальна змінна";

console.log(localVar); // Доступ до локальної змінної всередині функції

}

testLocalScope(); // Виведе: "Я локальна змінна"

console.log(localVar); // Спроба доступу до локальної змінної ззовні функції видасть помилку

У цьому прикладі, спроба доступу до localVar ззовні функції testLocalScope видасть помилку, оскільки localVar має локальну область видимості та доступна лише всередині функції.

### Підсумок

Розуміння областей видимості є критично важливим для ефективного управління змінними та запобігання помилкам у програмі. Глобальні змінні можуть бути доступні з будь-якого місця у вашому коді, але використання їх може призвести до конфліктів імен і ускладнити відладку. Локальні змінні допомагають уникнути цих проблем, забезпечуючи більшу модульність і читабельність коду.

## Ланцюжки областей видимості

Ланцюжки областей видимості в JavaScript — це концепція, що описує, як змінні та функції організовані та доступні в коді. Кожна функція, коли створюється, має доступ до змінних, що існують у її власній області видимості, а також до змінних з вищих областей видимості, аж до глобальної області. Це означає, що вкладені функції мають доступ не лише до власних локальних змінних, але й до змінних зовнішніх функцій, в яких вони були оголошені.

### Різні імена змінних

Приклад коду:

var globalVar = 'глобальна';

function outerFunction() {

var outerVar = 'зовнішня';

function innerFunction() {

var innerVar = 'внутрішня';

console.log(innerVar); // Виводить: 'внутрішня'

console.log(outerVar); // Виводить: 'зовнішня'

console.log(globalVar); // Виводить: 'глобальна'

}

innerFunction();

}

outerFunction();

У цьому прикладі, innerFunction є вкладеною у outerFunction. innerFunction має доступ до власної змінної innerVar, до змінної outerVar зовнішньої функції outerFunction, а також до глобальної змінної globalVar. Це ілюструє, як ланцюжок областей видимості дозволяє змінним і функціям взаємодіяти між собою в ієрархії коду.

Ланцюжки областей видимості забезпечують важливу основу для розуміння контексту виконання коду в JavaScript, дозволяючи розробникам створювати більш організовані та модульні програми.

### Однакові імена змінних

Ланцюжки областей видимості в JavaScript керують доступом до змінних, визначаючи, яка змінна з однаковим іменем буде використана у вкладених областях. Під час доступу до змінної, JavaScript шукає її спочатку у локальній області видимості функції, потім у батьківській, і так далі, до глобальної. Локальні змінні завжди мають пріоритет над зовнішніми з однаковими іменами.

Приклад з однаковими іменами змінних:

var someVar = 'глобальна';

function outerFunction() {

var someVar = 'зовнішня';

function innerFunction() {

var someVar = 'внутрішня';

console.log(someVar); // Виводить: 'внутрішня'

}

innerFunction();

console.log(someVar); // Виводить: 'зовнішня'

}

outerFunction();

console.log(someVar); // Виводить: 'глобальна'

У цьому прикладі, кожна функція має власну змінну someVar. Коли innerFunction викликається, вона використовує свою локальну змінну someVar ("внутрішня"), оскільки пошук змінної починається з найближчої області видимості. Аналогічно, коли console.log(someVar) викликається всередині outerFunction, використовується змінна someVar з області видимості outerFunction ("зовнішня"), а ззовні всіх функцій — глобальна змінна someVar ("глобальна").

Цей механізм ілюструє, як ланцюжки областей видимості забезпечують структурування та ізоляцію коду, дозволяючи розробникам створювати складні програми з чітким контролем над використанням змінних.

## Hoisting & Closure

### Hoisting (Підйом)

Hoisting, або підйом, це поведінка JavaScript, коли оголошення змінних та функцій "підіймаються" на початок їх області видимості перед виконанням коду. Це означає, що ми можемо викликати функції або звертатися до змінних до того, як вони були оголошені у коді.

### Приклад коду з функціями:

**Оголошення функції:**

console.log(sum(5, 5)); // Виведе: 10

function sum(a, b) {

return a + b;

}

У цьому прикладі, незважаючи на те, що функція sum оголошена після її виклику, код успішно працює та виводить результат. Підйом оголошення функції дозволяє нам викликати її перед її фізичним оголошенням у коді.

Коли функція оголошена як вираз функції через присвоєння змінній (з використанням var), hoisting поводиться інакше, ніж з оголошенням функції (function declaration). У цьому випадку, змінна, до якої присвоєно функцію, буде піднята, але саме присвоєння функції відбудеться в момент виконання коду, тому спроба викликати функцію перед її присвоєнням призведе до помилки.

**Функціональний вираз:**

console.log(myFunction); // Виведе: undefined

// myFunction(); // TypeError: myFunction is not a function

var myFunction = function() {

console.log("Це вираз функції, присвоєний змінній");

};

myFunction(); // Працює коректно після присвоєння

У цьому прикладі, коли ми спробували вивести myFunction в консоль перед присвоєнням функції, ми отримали undefined, оскільки оголошення змінної myFunction було піднято, але її значення на той момент було не визначено. Спроба викликати myFunction() перед присвоєнням функції призводить до помилки, оскільки змінна myFunction ще не має присвоєної функції і не вважається функцією. Після рядка з присвоєнням функції, виклик myFunction() працює коректно.

Ці приклади ілюструють важливість розуміння різниці між оголошенням функції та функціональним виразом у контексті hoisting, щоб уникнути помилок у виконанні програми.

### Особливості hoisting зі змінними:

Важливо зауважити, що hoisting поводиться по-різному з оголошеннями змінних. Оголошення змінної підіймається, але не її ініціалізація.

console.log(myVar); // Виведе: undefined

var myVar = 5;

console.log(myVar); // Виведе: 5

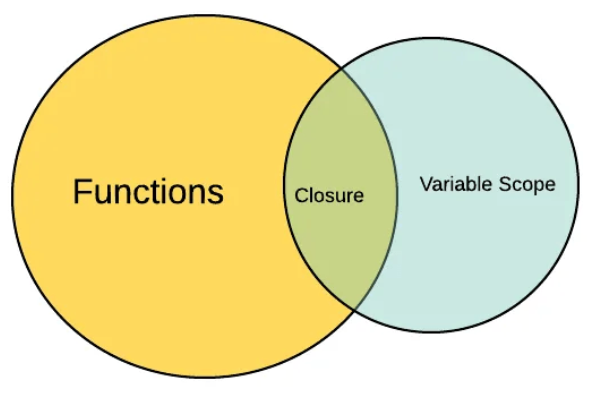
У цьому випадку, коли ми спробували доступитися до myVar перед її ініціалізацією, вона вже була оголошена завдяки підйому, тому не видає помилку, але має значення undefined.

Hoisting є важливою концепцією в JavaScript, яка впливає на те, як ми структуруємо наш код, і вимагає уважності, щоб уникнути потенційної плутанини при оголошенні змінних і функцій.

Error

### Closure (Замикання)

Замикання – це функція, що зберігає доступ до змінних зі свого лексичного середовища поза своєю первісною областю видимості, дозволяючи зберігати стан між викликами.



Приклад коду:

function createCounter() {

var count = 0; // Приватна змінна, доступна лише всередині createCounter

return function() {

count += 1; // Модифікуємо і повертаємо змінну count

return count;

};

}

const counter = createCounter(); // Створюємо екземпляр замикання

console.log(counter()); // Виведе: 1

console.log(counter()); // Виведе: 2

console.log(counter()); // Виведе: 3

У цьому прикладі, функція createCounter створює та повертає іншу функцію, яка має доступ до змінної count в області видимості createCounter. Кожен виклик counter() інкрементує та повертає змінну count, що демонструє як замикання дозволяє функціям "запам'ятовувати" стан і взаємодіяти з ним.

### ****Підсумок****

Коли інтерпретатор JavaScript виконує функцію, він створює об’єкт, який містить всі локальні змінні цього виклику функції. Цей об’єкт називається ‘**лексичним середовищем**’.

Кожна функція, визначена всередині цієї функції, має доступ до цього лексичного середовища через механізм, який називається ‘**областю видимості**’. Коли функція використовує змінні зі свого лексичного середовища після того, як зовнішня функція завершила виконання, ми називаємо це ‘**замиканням**’. Цей об’єкт з локальними змінними продовжує існувати, доки є хоча б одна функція, яка на нього посилається.

Це означає, що **замикання** в JavaScript - це спосіб зберігати приватні змінні, які існують після завершення виконання зовнішньої функції. Замикання зберігає посилання на ці змінні, створюючи приватну область видимості.

## Оголошення let та const

### Оголошення let

Ми використовуємо ключове слово let у JavaScript для оголошення змінних з областю видимості, обмеженою блоком коду, на відміну від var, що оголошує змінну з областю видимості, обмеженою функцією. Це означає, що змінні, оголошені за допомогою let, існують лише в рамках блоку {}, у якому вони були визначені, дозволяючи нам краще контролювати область видимості змінних і уникати помилок, пов'язаних із несподіваним доступом до них ззовні цього блоку.

Приклад коду:

function demoLet() {

let x = 10;

if (true) {

let x = 20; // Це інша 'x', оскільки 'let' обмежує область видимості блоком

console.log(x); // Виведе: 20

}

console.log(x); // Виведе: 10, оскільки змінна 'x', оголошена всередині блоку if, тут не доступна

}

demoLet();

У цьому прикладі ми демонструємо, як let дозволяє оголошувати змінні, що мають область видимості, обмежену конкретним блоком коду. Змінна x, оголошена всередині функції demoLet, має значення 10, але всередині блоку if, ми оголошуємо іншу змінну x з областю видимості, обмеженою цим блоком, і присвоюємо їй значення 20. Це показує, як змінні з однаковими іменами, але оголошені в різних областях видимості за допомогою let, можуть існувати незалежно одна від одної.

### Пояснення правил області видимості для let:

1. **Область видимості змінної let – блок {...}**: Це означає, що змінні, оголошені за допомогою let, доступні лише в рамках блоку, у якому вони були визначені, та в усіх вкладених блоках.
2. **Змінна let не існує до її оголошення у своїй області видимості**: Це відомо як "тимчасова мертва зона", що означає, що доступ до змінної let перед її оголошенням у блоку призведе до помилки ReferenceError.
3. **Повторне оголошення let змінних у тому ж скоупі викликає помилку:** Спроба повторного оголошення змінної призведе до синтаксичної помилки.
4. **При використанні в циклі для кожної ітерації створюється своя змінна**: Коли let використовується в умовах циклу, для кожного проходу циклу створюється нова змінна, що дозволяє зберігати унікальний стан змінної для кожної ітерації.

### Приклад коду з описом:

function demoScope() {

// Приклад 1: Область видимості змінної `let` – блок `{...}`

if (true) {

let blockScoped = "Доступна всередині цього блоку";

console.log(blockScoped); // Виведе: Доступна всередині цього блоку

}

// console.log(blockScoped); // Помилка: blockScoped не визначена

// Приклад 2: Повторне оголошення let змінних у тому ж скоупі

let x;

// let x; // помилка: змінна x вже оголошена

// Приклад 3: "Тимчасова мертва зона"

// console.log(tempDeadZone); // Помилка: Cannot access 'tempDeadZone' before initialization

let tempDeadZone = "Ініціалізована після використання";

// Приклад 4: Унікальна змінна для кожної ітерації циклу

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

setTimeout(function() { console.log(i); }, 1000); // Виведе 0, потім 1, потім 2

}

}

demoScope();

У цьому прикладі:

* **Приклад 1** ілюструє, як змінна blockScoped, оголошена в блоку if, доступна лише всередині цього блоку.
* **Приклад 2** показує що спроба повторного оголошення змінної призведе до синтаксичної помилки.
* **Приклад 3** демонструє "тимчасову мертву зону", показуючи, що змінна tempDeadZone не може бути використана до її оголошення.
* **Приклад 4** показує, як при кожній ітерації циклу for створюється нова змінна i, що дозволяє setTimeout вивести унікальне значення i для кожної ітерації.

### Оголошення const

Ми використовуємо оголошення const для створення змінних, значення яких не можна змінювати після їх ініціалізації. Змінні, оголошені за допомогою const, мають область видимості блоку подібно до let і повинні бути ініціалізовані при оголошенні.

Приклад коду:

const PI = 3.14; // Оголошуємо константу PI та присвоюємо їй значення

console.log(PI); // Виводимо значення PI

// Спроба змінити значення PI призведе до помилки

// PI = 3.14159; // TypeError: Assignment to constant variable.

У цьому прикладі ми оголошуємо константу PI та присвоюємо їй значення 3.14. Спроба змінити значення PI викличе помилку, оскільки змінні, оголошені за допомогою const, не можуть бути переприсвоєні після їх ініціалізації. Це демонструє ключову властивість const - незмінність значення після оголошення.

### Коли використовувати camelCase та SNAKE\_CASE для констант в JavaScript

У JavaScript, вибір між camelCase та SNAKE\_CASE для констант залежить від контексту їх використання:

* **camelCase** використовують для локальних констант або коли константа є об'єктом, що може змінюватися (наприклад, об'єкти, масиви), але посилання на який залишається незмінним.
* **SNAKE\_CASE** застосовують для глобальних констант або констант на рівні модуля, значення яких є справді незмінними та важливими для всього додатку.

### Приклад коду:

// camelCase для локальних констант або змінних об'єктів

const config = {

apiKey: "ABC123",

};

// SNAKE\_CASE для глобальних незмінних констант

const MAX\_USERS = 100;

function checkUsers(users) {

if (users.length > MAX\_USERS) {

console.log("Перевищено максимальну кількість користувачів");

}

}

console.log(config.apiKey); // Використання константи об'єкта

checkUsers([1, 2, 3]); // Приклад використання глобальної константи

У цьому прикладі config є локальною константою, що використовує camelCase, оскільки вона відноситься до об'єкта конфігурації. Натомість, MAX\_USERS визначено в SNAKE\_CASE, бо це глобальна константа, що має незмінне значення в контексті всього додатку.

**Scope (розширене розуміння областей видимості з let та const)**

Область видимості в JavaScript визначає, де змінні, константи та функції можуть бути доступні та використані у коді. Розрізняють кілька типів областей видимості:

* **Глобальна область видимості**: Елементи, оголошені на найвищому рівні коду, доступні з будь-якого місця у програмі.
* **Локальна область видимості**: Включає в себе:
* **Функціональна область видимості (для** var**)**: Змінні, оголошені всередині функцій за допомогою **var**, доступні лише в межах цієї функції та її вкладених функцій.
* **Блокова область видимості**: Елементи, оголошені за допомогою let та const всередині блоків коду (наприклад, у циклах або умовних операторах {...}), доступні лише в межах цього блоку. Це відрізняється від змінних, оголошених через var, які мають функціональну область видимості або стають глобальними, якщо оголошені поза функціями.

Коли let та const використовуються всередині функції, вони створюють локальну область видимості для цієї функції, але з додатковою особливістю блокової області видимості. Це означає, що змінні, оголошені за допомогою let та const всередині функції, будуть локальними для цієї функції, але вони також обмежені будь-якими блоками ({}), у яких були оголошені.

Приклад для ілюстрації:

function testFunction() {

let x = 10; // Локальна для testFunction

if (true) {

let y = 20; // Блокова область видимості, локальна для блоку if

console.log(x); // Доступно, бо x локальна для функції

console.log(y); // Доступно, бо y локальна для блоку if

}

console.log(x); // Доступно, бо x локальна для функції

// console.log(y); // Помилка, y не доступна за межами блоку if

}

testFunction();

У цьому прикладі x є локальною змінною для testFunction завдяки let, а y має блокову область видимості всередині умовного блоку if і не доступна за його межами. Таким чином, хоча let та const і створюють локальну область видимості всередині функції, їхня унікальна властивість полягає у створенні блокової області видимості в межах будь-яких блоків коду всередині цієї функції.

### Відмінності для let та const:

* Як let, так і const дозволяють створювати змінні з блоковою областю видимості, обмежуючи їх використання рамками блоку {}, де вони були оголошені.
* Змінні, оголошені через let та const, не піднімаються (hoisting) на початок функції або блоку коду, в якому вони оголошені, на відміну від var. Це може призвести до ReferenceError, якщо спробувати використати змінну до її оголошення.
* Повторне оголошення змінної в тій же області видимості з let або const призведе до синтаксичної помилки, що запобігає неявному перепризначенню змінних.
* Основна відмінність між let та const полягає в тому, що змінні, оголошені за допомогою const, мають бути ініціалізовані в момент оголошення та не можуть бути змінені після цього. Це робить const ідеальним для оголошення констант, значення яких не повинно змінюватися.

Ці особливості роблять let та const ідеальними виборами для управління областю видимості змінних в сучасних JavaScript-додатках, надаючи більше контролю та зменшуючи потенційні помилки в коді, при цьому const забезпечує додаткову гарантію незмінності для констант.

## Частина 1

### Використовуйте виразні, зрозумілі назви змінних

Погано

var n = 3.14159; // Незрозуміло, що означає 'n'

var c = 'John Doe'; // Неясно, що саме ця змінна представляє

var d = new Date().toISOString(); // Неочевидно, до чого стосується 'd'

Добре

const PI\_VALUE = 3.14159; // Зрозуміло, що це математична константа пі

const userName = 'John Doe'; // Зрозуміло, що це ім'я користувача

const currentDateISO = new Date().toISOString(); // Явно вказує на поточну дату в форматі ISO

У "Погано" прикладах, імена змінних не дають чіткого розуміння їхнього вмісту або призначення, тоді як у "Добре" прикладах, імена змінних явно описують, що вони зберігають або представляють, роблячи код легшим для розуміння і підтримки.

Error

### Не використовуйте var

Погано

var price = 100; // var не використовує блочну область видимості, що може викликати помилки

var userLoggedIn = true; // Знову використання var може призвести до перезапису змінної в майбутньому

Добре

let price = 100; // let має блочну область видимості і дозволяє змінювати значення

const userLoggedIn = true; // const запобігає переприсвоєнню, використовуйте для незмінних значень

### Використовуйте імена, зручні для пошуку

Погано

setTimeout(tick, 86400000); // Незрозуміло, що означає число 86400000

Добре

const MILLISECONDS\_IN\_A\_DAY = 86400000;

setTimeout(tick, MILLISECONDS\_IN\_A\_DAY); // Чітко і зрозуміло,

### Явність краща за неявність

Погано

const locations = ['Austin', 'New York', 'San Francisco'];

locations.forEach((l) => {

doStuff();

doSomeOtherStuff();

// ...

// ...

// Стійте. Ще раз, що таке 'l'?

dispatch(l);

});

У цьому фрагменті коду використовується l як параметр функції, що може бути неясним, особливо без контексту.

**Добре**

const locations = ['Austin', 'New York', 'San Francisco'];

locations.forEach((location) => {

doStuff();

doSomeOtherStuff();

// ...

// ...

// ...

dispatch(location);

});

У цьому фрагменті параметр location ясно позначає, що він представляє елемент масиву locations, що робить код більш зрозумілим.

Error

### Не додавайте зайвий контекст

Погано

const Car = {

carMake: 'Honda',

carModel: 'Accord',

carColor: 'Blue',

};

function paintCar(car) {

car.carColor = 'Red';

}

Тут префікс car у властивостях об'єкта Car є зайвим, оскільки вже зрозуміло з контексту, що властивості стосуються автомобіля.

**Добре**

const Car = {

make: 'Honda',

model: 'Accord',

color: 'Blue',

};

function paintCar(car) {

car.color = 'Red';

}

У цьому прикладі зайвий контекст в назвах властивостей видалено, що робить код чистішим та зрозумілішим.

Error

### Використовуйте аргументи за замовчуванням

Погано

function createMicrobrewery(name) {

const breweryName = name || 'Hipster Brew Co.';

// ...

}

У цьому прикладі для встановлення значення за замовчуванням використовується оператор ||. Однак, якщо name буде дорівнювати будь-якому "falsy" значенню, такому як пустий рядок, 0, false, null, чи NaN, то буде використане 'Hipster Brew Co.'.

**Добре**

function createMicrobrewery(breweryName = 'Hipster Brew Co.') {

// ...

}

У цьому фрагменті використовується синтаксис аргументів за замовчуванням ES6, що є чистішим та більш надійним способом встановлення значень за замовчуванням, оскільки воно застосовується тільки коли breweryName не передано або undefined.

## Частина 2

### Аргументи функції, не більше трьох

Погано

function createMenu(title, body, buttonText, cancellable) {

// ...

}

Цей приклад показує функцію з чотирма параметрами, що робить її виклик і тестування більш складними.

**Добре**

createMenu({

title: 'Foo',

body: 'Bar',

buttonText: 'Baz',

cancellable: true,

});

У цьому прикладі параметри функції згруповані в один об'єкт, що робить функцію більш гнучкою та читабельною. Також це полегшує передачу параметрів і дозволяє встановлювати необов'язкові параметри без строгої послідовності.

Error

### Функції повинні виконувати одну дію

Погано

function emailClients(clients) {

clients.forEach((client) => {

const clientRecord = database.lookup(client);

if (clientRecord.isActive()) {

email(client);

}

});

}

У цій функції виконуються декілька дій: пошук запису клієнта в базі даних, перевірка активності клієнта та відправлення електронної пошти.

Це робить функцію більш складною і менш зрозумілою.

**Добре**

function emailClients(clients) {

clients.filter(isClientActive).forEach(email);

}

function isClientActive(client) {

const clientRecord = database.lookup(client);

return clientRecord.isActive();

}

Тут логіка визначення активності клієнта винесена в окрему функцію isClientActive, а сама функція emailClients виконує лише дві дії: фільтрацію активних клієнтів і відправлення їм електронної пошти.

Кожна дія відповідає одній відповідальності, що є прикладом ясного та чистого коду.

### Назви функцій мають відображати їх дії

Погано

function addToDate(date, month) {

// ...

}

const date = new Date();

// Из имени функции сложно понять, что она добавляет

addToDate(date, 1);

У цьому прикладі назва функції addToDate не дає чіткого розуміння того, що саме додається до дати.

**Добре**

function addMonthToDate(month, date) {

// ...

}

const date = new Date();

addMonthToDate(1, date);

У цьому прикладі назва функції addMonthToDate явно вказує, що до дати додається місяць, що робить наміри коду більш зрозумілими.

### ****Функції повинні мати один рівень абстракції****

Погано

function parseBetterJSAlternative(code) {

const REGEXES = [

/\* ... \*/

]

const statements = code.split(';')

const tokens = []

REGEXES.forEach((REGEX) => {

statements.forEach((statement) => {

/\* ... \*/

}) // Розбиття коду на речення

})

tokens.forEach((token) => {

// Токенізація коду

})

ast.forEach((node) => {

// Побудова AST }) }

Функція parseBetterJSAlternative включає кілька різних операцій на різних рівнях абстракції: розбивання коду на речення, токенізацію і аналіз синтаксичного дерева (AST), що робить її складною для розуміння та тестування.

**Добре**

function tokenize(code) { /\* Токенізація коду \*/ }

function buildAST(tokens) { /\* Побудова AST \*/ }

function parseBetterJSAlternative(code) {

const tokens = tokenize(code); // Виклик функції токенізації

const ast = buildAST(tokens); // Виклик функції побудови AST

// Розбиття коду на речення

}

Функції tokenize, buildAST, і parseBetterJSAlternative поділяють логіку на менші, більш зрозумілі частини, кожна з яких виконує одну конкретну задачу.

Цей підхід дозволяє кожній функції працювати на одному рівні абстракції, спрощує їх тестування та підтримку.

### ****Уникайте дублювання коду****

Погано

function showDeveloperList(developers) {

developers.forEach((developer) => {

const expectedSalary = developer.calculateExpectedSalary()

const experience = developer.getExperience()

const githubLink = developer.getGithubLink()

const data = { expectedSalary, experience, githubLink }

render(data)

})

}

function showManagerList(managers) {

managers.forEach((manager) => {

const expectedSalary = manager.calculateExpectedSalary()

const experience = manager.getExperience()

const portfolio = manager.getMBAPprojects()

const data = { expectedSalary, experience, portfolio }

render(data)

})

}

Тут код для створення даних, які передаються в render, повторюється в обох функціях.

**Добре**

function showEmployeeList(employees) {

employees.forEach((employee) => {

const expectedSalary = employee.calculateExpectedSalary();

const experience = employee.getExperience();

let portfolio = employee.getGithubLink();

if (employee.type === 'manager') {

portfolio = employee.getMBAPprojects();

}

const data = { expectedSalary, experience, portfolio };

render(data);

});

}

У вдосконаленому варіанті функція showEmployeeList обробляє список працівників універсально, використовуючи умову для визначення, які дані потрібно додати в портфоліо, тим самим усуваючи дублювання коду.

### Не використовуйте флаги як параметри функції

Погано

function createFile(name, temp) {

if (temp) {

fs.create(`./temp/${name}`);

} else {

fs.create(name);

}

}

Функція використовує флаг temp для визначення шляху створення файлу, що робить її відповідальною за два різних варіанти поведінки.

**Добре**

function createFile(name) {

fs.create(name);

}

function createTempFile(name) {

createFile(`./temp/${name}`);

}

Розділені функції createFile і createTempFile кожна відповідає за один шлях виконання, що забезпечує більш чистий та зрозумілий код.

Цей підхід дозволяє легше читати і розуміти код, а також полегшує рефакторинг і тестування окремих функцій.

## Частина 3

### Уникайте побічних ефектів

Погано

let name = 'John Smith';

function splitIntoFirstAndLastName() {

name = name.split(' ');

}

splitIntoFirstAndLastName();

console.log(name); // ['John', 'Smith']

Функція splitIntoFirstAndLastName змінює глобальну змінну name, що є побічним ефектом.

**Добре**

function splitIntoFirstAndLastName(name) {

return name.split(' ');

}

const name = 'John Smith';

const newName = splitIntoFirstAndLastName(name);

console.log(name); // 'John Smith'

console.log(newName); // ['John', 'Smith']

Функція splitIntoFirstAndLastName приймає параметр і повертає нове значення, не змінюючи зовнішній стан.

У "доброму" прикладі функція не має побічних ефектів, оскільки вона не змінює ніяких зовнішніх змінних і повертає нове значення, замість того, щоб змінювати вхідний параметр.

Error

### ****Перевага функціонального програмування****

Погано

const programmerOutput = [

{ name: 'John Smith', linesOfCode: 500 },

// ...

];

let totalOutput = 0;

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

totalOutput += programmerOutput[i].linesOfCode;

}

Погано (Імперативний стиль):

В імперативному стилі використовується цикл for для обчислення сумарної кількості рядків коду виведених програмістами.

**Добре**

const programmerOutput = [

{ name: 'John Smith', linesOfCode: 500 },

// ...

];

const INITIAL\_VALUE = 0;

const totalOutput = programmerOutput

.map((programmer) => programmer.linesOfCode)

.reduce((acc, linesOfCode) => acc + linesOfCode, INITIAL\_VALUE);

Добре (Функціональний стиль):

У функціональному стилі застосовуються методи map та reduce для отримання того ж результату. Це забезпечує більш декларативний підхід, що полегшує читання та зрозумілість коду.

У "доброму" прикладі використання функціонального стилю дозволяє написати код, що легше масштабується та змінюється, оскільки він не залежить від зовнішнього стану і має менше побічних ефектів.

### ****Уникайте негативних умов****

Погано

function isDOMnodeNotPresent(node) {

// ...

}

if (!isDOMnodeNotPresent(node)) {

// ...

}

Негативна назва функції isDOMnodeNotPresent може зробити код менш читабельним, особливо коли використовується з запереченням, що призводить до подвійного заперечення, яке ускладнює розуміння.

**Добре**

function isDOMnodePresent(node) {

// ...

}

if (isDOMnodePresent(node)) {

// ...

}

Позитивна назва функції isDOMnodePresent робить логіку коду більш прямолінійною та зрозумілою.

У "доброму" прикладі умова читається як "якщо DOM вузол присутній", що є більш інтуїтивно зрозумілим.

## Частина 4

### ****Видаляйте невикористаний код****

Погано

function oldRequestModule(url) {

// ...

}

function newRequestModule(url) {

// ...

}

const req = newRequestModule;

inventoryTracker('android', req, 'www.inventory-awesome.io');

В коді присутня застаріла функція oldRequestModule, яка більше не використовується.

**Добре**

function newRequestModule(url) {

// ...

}

const req = newRequestModule;

inventoryTracker('android', req, 'www.inventory-awesome.io');

Застаріла функція видалена, залишається тільки актуальний код.

У "доброму" прикладі код більш чистий і легший для підтримки, оскільки в ньому відсутній невикористаний код, який може заплутати розробників або призвести до помилок.

### ****Не залишайте закоментований код****

Погано

doStuff();

// doOtherStuff();

// doSomeMoreStuff();

// doSoMuchStuff();

Код містить закоментовані виклики функцій, що може заплутати розробників або призвести до непорозумінь щодо того, чи ці рядки мають бути в коді, чи ні.

**Добре**

doStuff();

Усі закоментовані виклики функцій видалені, залишаючи тільки активний код, що покращує читабельність і зменшує зайвий шум у коді.

Ця практика підтримує чистоту кодової бази і використання систем контролю версій для зберігання історії змін, замість того, щоб залишати застарілий або невикористаний код у коментарях.

### ****Уникайте маркерів візуального розділення коду****

Погано

/////////////////////////////////////////////////////////////////////////////////

// Створення об'єкта

/////////////////////////////////////////////////////////////////////////////////

$scope.model = {

menu: 'foo',

nav: 'bar',

};

/////////////////////////////////////////////////////////////////////////////////

// Встановлення змінної

/////////////////////////////////////////////////////////////////////////////////

const actions = function () {

// ...

};

Використання рядків з символами для розділення блоків коду, що створює візуальний "шум" і може відволікати від змісту коду.

**Добре**

$scope.model = {

menu: 'foo',

nav: 'bar',

};

const actions = function () {

// ...

};

Чистий код з правильними відступами і форматуванням, без зайвих маркерів, який є більш читабельним.

У "доброму" прикладі код легше читати, і його структура зрозуміла без додаткових візуальних розділювачів.

## Recursion (Рекурсія)

Рекурсія — це процес, у якому функція викликає сама себе як частину свого виконання. Ми використовуємо рекурсію для розв'язання задач, що можуть бути розбиті на подібні підзадачі меншого розміру, і часто вона використовується для роботи з даними, які мають вкладену структуру, наприклад, дерева або списки.

Приклад коду функції для обчислення факторіалу числа за допомогою рекурсії:

function factorial(n) {

// Базовий випадок

if (n === 0 || n === 1) {

return 1;

}

// Рекурсивний випадок

return n \* factorial(n - 1);

}

console.log(factorial(5)); // Виведе 120, оскільки 5! = 5 \* 4 \* 3 \* 2 \* 1 = 120

У цьому прикладі, factorial є рекурсивною функцією, яка обчислює факторіал числа n. Якщо n дорівнює 0 або 1, функція повертає 1 (базовий випадок, що зупиняє рекурсію), інакше вона повертає n помножене на факторіал n-1 (рекурсивний випадок).