

IIFE. HoF. Currying. Scope. Hoisting.
Closures. Let. Const. Best
practices.
Recursions

Зміст уроку

1. [Повторюємо callback функції](#)
2. [Повернення функції із функцій](#)
3. [Immediately Invoked Function Expression \(IIFE\)](#)
4. [Higher-Order Functions \(HoF\)](#)
5. [Currying \(Карпування, Карпінг\)](#)
6. [Basic Scope \(області видимості змінних з var\)](#)
7. [Hoisting \(Підйом\)](#)
8. [Closure \(Замикання\)](#)
9. [Оголошення let](#)
10. [Пояснення правил області видимості для let](#)
11. [Оголошення const](#)
12. [Коли використовувати camelCase та SNAKE_CASE для констант в JavaScript](#)
13. [Scope \(розширене розуміння областей видимості з let та const\)](#)
14. [Best practices](#)
15. [Рекурсія](#)

Quiz

1. Який оператор використовується для явного перетворення рядка в число?
a. `String()`
b. `ParseInt()`
c. `Number()`
d. `ToNumber()`
2. Яка конструкція використовується для виконання коду, якщо певна умова істинна?
a. `switch`
b. `if {}`
c. `for`
d. `while`
3. Який вираз використовується для виконання одного блоку коду замість іншого на основі умови?
a. `switch`
b. `for`
c. `if ... else`
d. `while`
4. Яка конструкція дозволяє виконати різні дії на основі різних умов?
a. `for`
b. `while`
c. `do...while`
d. `if ... else if ... else`
5. Яка конструкція використовується для вибору одного з багатьох блоків коду для виконання?
a. `if`
b. `switch ... case`
c. `for`
d. `while`
6. Який оператор використовується для виконання кількох виразів у одному рядку коду?
a. `&&`
b. `||`
c. `,` (оператор кома)
d. `:`
7. Через який цикл можна перебрати властивості об'єкта?
a. `for`
b. `for...in`
c. `for...of`
d. `while`
8. Через який цикл можна перебрати ітеровані об'єкти, такі як масиви і рядки?
a. `for`
b. `for...in`
c. `for...of`
d. `while`
9. Яка функція викликається іншою функцією як аргумент?
a. Анонімна функція
b. Callback функція
c. Замикання (Closure)
d. IIFE (Immediately Invoked Function Expression)
10. Який оператор дозволяє "вийти" з циклу, коли виконується певна умова?
a. `continue`
b. `break`
c. `return`
d. `exit`

Повторюємо callback
функці

Callback функці

Callback функція в JavaScript - це функція, яку ми передаємо як аргумент іншій функції .

Callback функція є основним механізмом для забезпечення асинхронності в JavaScript

Синхронний код

```
function first() {  
  console.log('Це функція first');  
}
```

```
function second() {  
  console.log('Це функція second');  
}
```

// Виклик функцій

```
first();  
second();
```

// Використовуємо callback

```
function first(callback) {  
  console.log('Це функція first');  
  callback();  
}
```

```
function second() {  
  console.log('Це функція second');  
}
```

// Виклик функцій

```
first(second);
```

Асинхронний код

```
function first() {  
  setTimeout(function () {  
    console.log('Це функція  
  }, 5000); // Затримка 5 секунд  
}
```

```
function second() {  
  setTimeout(function () {  
    console.log('Це функція  
  }, 2000); // Затримка 2 секунди  
}
```

// Виклик функцій

```
first();  
second();
```

```
function first(callback) {  
  setTimeout(function () {  
    console.log('Це функція  
    callback(); // Виклик callback-функції  
  }, 5000); // Затримка 5 секунд  
}
```

```
function second() {  
  setTimeout(function () {  
    console.log('Це функція  
  }, 2000); // Затримка 2 секунди  
}
```

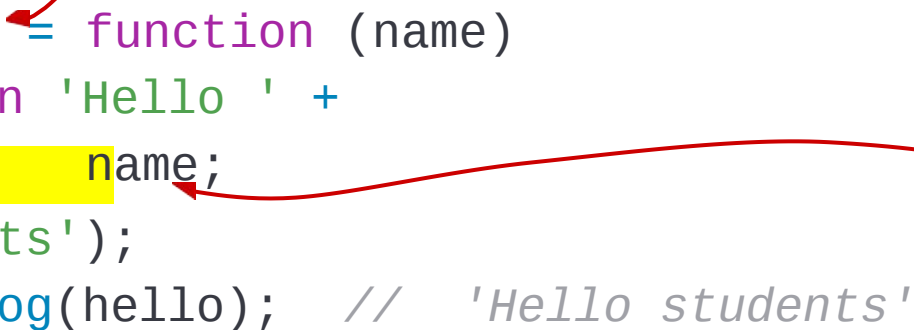
// Виклик функції first, передаючи second як callback

```
first(second);
```


Повернення функці із
функцій

Функція може бути викликана одразу після визначення, якщо ми використовуємо **вираз визначення**.

```
var hello = function (name)
  { return 'Hello ' +
    name;
  }('students');
console.log(hello);  // 'Hello students'
```



Після цього виразу визначення ми можемо поставити **круглі дужки** з аргументами і таким чином отримаємо **вираз виклику**.

Подивившись першу частину, можна подумати, що ми присволи зміній функцію, хоча насправді ми присвоюємо їй значення функції.

```
var hello = function (name) {...}('students');
```

Для більшої ясності у таких випадках вираз виклику беруть у круглі дужки:

```
var hello = (function(name) {...}('students'));
```

A red curved arrow points from the text "вираз виклику беруть у круглі дужки:" to the parentheses in the code snippet below. Two yellow squares highlight the opening and closing parentheses of the function call expression in the code: `(function(name) {...}('students'))`.

Immediately Invoked Function Expression (IIFE)

IIFE, або **Immediately Invoked Function Expression**, це конструкція в JavaScript, яка дозволяє виконати функцію негайно після оголошення. Це корисно для створення "приватних" змінних та функцій, які не впливають на глобальний об'єкт.

```
(function() {  
    var privateVariable = "Приватна змінна";  
    console.log(privateVariable);  
})();
```

Higher-Order Functions (HoF)

Higher-order function

Функція вищого порядку (Higher-order function) - це функція, яка приймає одну або кілька функцій як аргументи, виконує деякі операції, а потім повертає результат. Вони є основою функціонального програмування в JavaScript.

```
const apply = (x, fn) => fn(x);  
const double = x => x * 2;  
const result1 = apply(3, double);  
const result2 = apply(5, x => x + 2);
```

скорочений
запис через
const та
функції стрілки

```
var apply = function(x, fn) {  
    return fn(x);  
};
```

```
var double = function(x) {  
    return x * 2;  
};
```

```
var result1 = apply(3, double);
```

```
var result2 = apply(5, function(x) {  
    return x + 2;  
});
```

попередній приклад
через **var** та **function**

Currying
(Каррування, Каррінг)

Каррування або **каррінг** (currying) - це техніка в програмуванні, коли ми перетворюємо функцію з кількома аргументами на послідовність функцій, кожна з яких приймає лише один аргумент.

Цей підхід дозволяє нам створювати більш гнучкі функції, що можуть бути адаптовані під різні сценарії.

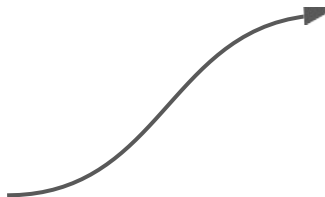
Звичайна функція для додавання двох чисел:

```
function add(a, b) {  
  return a + b;  
}
```



За допомогою каррування, ми можемо перетворити цю функцію на послідовність функцій, де кожна приймає один аргумент:

```
function curryAdd(a) {  
  return function (b) {  
    return a + b;  
  };  
}
```



Тепер ми можемо використовувати **curryAdd** так:

```
// Функція, що додає 5  
let addFive = curryAdd(5);  
  
// Виведе 15  
console.log(addFive(10));
```

Basic Scope

(області видимості змінних з
var)

Область видимості змінно для var

Область видимості змінної – це частина програми, де ця змінна визначена та доступна.

Змінні області видимості діляться на **глобальні** і **локальні**.

Глобальними називаються всі змінні, оголошені поза будь-якими функціями. Змінні, оголошені всередині функцій, є **локальними**.

Локальна змінна з таким самим ім'ям як і глобальна має більший пріоритет.

Можемо переконатись у цьому на наступному прикладі:

```
var i = 5;  
var func =function ()  
    { var i =10;  
      console.log(i);  
    };  
func();           // 10  
console.log(i);   // 5
```

Ланцюжки областей видимості

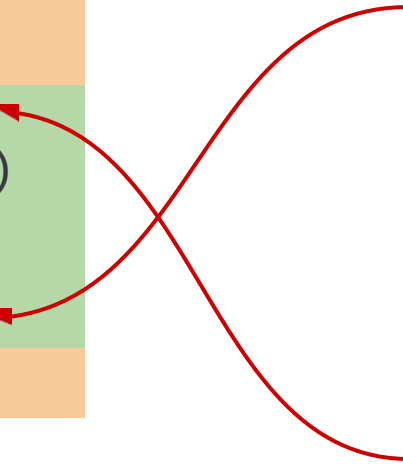
Ланцюжки областей видимості в JavaScript — це концепція, що описує, як змінні та функції організовані та доступні в коді.

Кожна функція, коли створюється, має доступ до змінних, що існують у 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.

```
var i = 5;  
var func = function ()  
{ var i = 10;  
  console.log(i);    // 10  
  var innerFunc = function ()  
  { var i = 15;  
    console.log(i);  // 15  
  }  
  innerFunc();  
};  
func();
```

Коли звертаємося до змінної `i`, інтерпретатор передусім перевіряє **першу область видимості в ланцюжку**, тобто. Функцію змінної **`innerFunc`**. Якщо для цієї функції не виявиться змінної, то інтерпретатор спробує знайти цю змінну у **наступній області видимості в ланцюжку**, тобто у змінній **`func`**.



Ми можемо перекоонатися в цьому, прибравши оголошення змінно **var i = 15**.

```
var i = 5;
var func =function ()
{
  var i =10;
  console.log(i);    // 10
  var innerFunc = function ()
  {
    console.log(i); // 10
  }
};innerFunc();
func();
```

Хоча при оголошенні глобальних змінних можна не використовувати ключове слово **var** і це не вплине на визначення змінно **в глобальній області видимості**, у строгому режимі **'use strict'** це призведе до помилки.

Для **локальних змінних у функціях** необхідно використовувати **var** або хні аналоги **let** та **const** в ES6, інакше змінна може ненавмисно стати глобальною і потенційно змінити змінну вищо області видимості, якщо така існує з таким самим іменем.

Hoisting
(Підйом)

Hoisting, або **підйом**, це поведінка JavaScript, коли оголошення змінних та функцій "підіймаються" на початок 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 **неправильний**:

```
console.log(i); // undefined  
var i = 15;
```

Приклад **правильний**:

```
var i = 15;  
console.log(i); // 15
```

Буде **undefined**, оскільки **var i** буде піднято, але значення ще не присвоєно

```
console.log(i); // undefined
```

```
var i = 15;
```

Буде **undefined**, оскільки **var i** буде піднято, але значення ще не присвоєно

```
var i // i === undefined
```

```
console.log(i) // undefined
```

```
i = 15 // i === 15
```

Оголошення змінної *i* без ініціалізації, *i* отримує значення **undefined**

Присвоєння змінній *i* значення 15

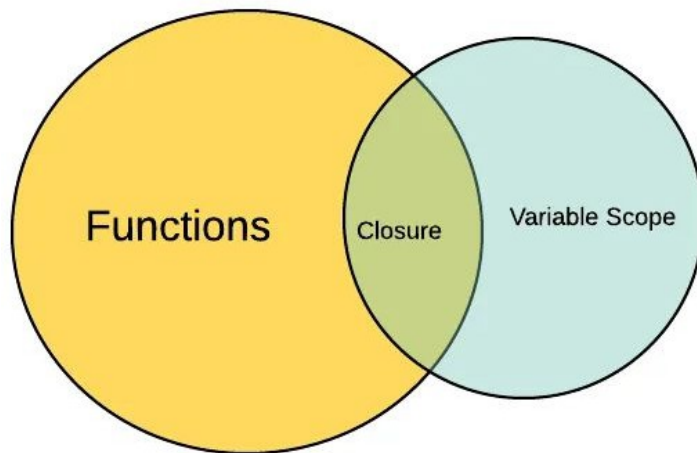
Виводиться **undefined**, оскільки *i* ще не має присвоєного значення

У JavaScript, **var** оголошення піднімаються на початок функції без ініціалізації. Тому, якщо ви спробуєте вивести змінну до ініціалізації, ви отримаєте **undefined**.

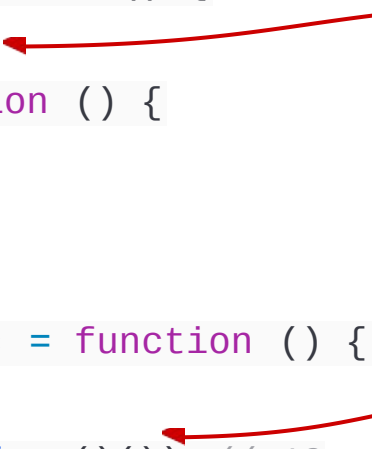
Щоб уникнути цього, **рекомендується оголошувати змінні на початку області видимості.**

Closure
(Замикання)

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



```
var func = function () {  
  var i = 10  
  return function () {  
    return i  
  }  
}  
  
var anotherFunc = function () {  
  var i = 20  
  console.log(func()) // 10  
}  
  
anotherFunc()
```



Оголосимо функцію, всередині якої **ініціалізуємо** `i = 10` і повернемо з цієї функції функцію, яка буде повертати значення змінної `i`.

Тепер оголосимо ще одну функцію, всередині якої оголосимо змінну `i = 20` і **виклинемо** тут функцію, яка повертається з функції `func` і виведемо значення, що повертається.

Через лексичну область видимості виводиться змінна `i`, оголошена у **func**, а не в **anotherFunc**.

```
var func = function () {  
    var i = 10  
    return function () {  
        return i  
    }  
}  
var myFunc = func()  
var anotherFunc = function () {  
    var i = 20  
    console.log(myFunc()) // 10  
}  
anotherFunc()
```

Для ще більшо наочності можна **присвоїти** змінній функцію, що повертається з функції **func** і потім **викликати** функцію тут.

Результат при цьому не змінити.

Кожна функція в JavaScript **може стати замиканням**, якщо вона використовує змінні зі свого лексичного середовища після того, як зовнішня функція завершила виконання. **Замикання зберігає доступ до цих змінних**, створюючи приватний область видимості.

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.

У нашій функції ми повертатимемо значення змінної **count**, збільшене на одиницю і в цьому замиканні ініціалізуємо змінну **count**.

```
var counter = (function ()  
  { var count = 0;  
    return function ()  
      { return ++count;  
      }  
  }  
)();
```

	Elements	Console	Sources	>>	:	X
	top	Filter	Default levels	▼	1 iter	
0		main.js:8				
1		main.js:9				
2		main.js:10				
3		main.js:11				
4		main.js:12				
5		main.js:13				
>						

Таким чином ми створили функцію, яка не приймає жодних аргументів і при цьому повертає різні значення за різних викликів.

Трохи покращимо нашу функцію, додавши можливість зміни значення лічильника.

Ця функція прийматиме якесь число (**num**) і якщо цей аргумент було передано, тобто. якщо змінна **num !== undefined**, ми присвоюємо лічильнику це число, інакше значення залишається тим же.

<https://jsfiddle.net/fomenkoandrey/bL5ht9rx/32/>

Варто сказати, що замикання – не єдиний спосіб, щоб досягти тако поведінки.

Оскільки функції є об'єктами, ми можемо додати змінну **count** як властивість об'єкта **counter** і всередині функції змінювати цю властивість. В результаті отримаємо таку ж поведінку, але при іншій реалізації . [Приклад...](#)



Object like solution...

Closure / Замикання



Оголошення let

Ключове слово **let** у JavaScript використовується для оголошення змінних з областю видимості, обмеженою блоком коду **{}**, на відміну від **var**, яке має область видимості, обмежену функцією або глобальним контекстом. Це означає, що змінні, оголошені за допомогою **let**, доступні лише в межах блоку, де вони визначені, що забезпечує кращий контроль за хньою видимістю та запобігає доступу до них поза цим блоком, знижуючи ймовірність виникнення помилок.

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 оголошенням у блоку призведе до помилки **ReferenceError**.

3. Повторне оголошення **let** змінних у тому ж скоупі викликає помилку:

Спроба **повторного оголошення змінної** призведе до синтаксично помилки.

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

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 змінно для кожної ітерації.

Змінна **var** – одна на всі ітерації циклу і видно навіть після циклу:

```
for(var i=0; i<10; i++) { /* ... */ }  
console.log(i);    // 10
```

У циклах кожна ітерація має унікальну **let** змінну.

```
for(let i=0; i<10; i++) { /* ... */ }  
console.log(i);    // undefined
```


Наприклад, два таких цикли не конфліктують:

```
// кожен цикл має свою змінну i
for (let i = 0; i < 10; i++) { /* ... */ }
for (let i = 0; i < 10; i++) { /* ... */ }
console.log(i) // помилка: глобальної i немає
```

Змінна **i**, оголошена в циклі через **let**, доступна тільки у цьому циклі; спроба доступу ззовні викличе помилку,

Оголошення const

Оголошення **const** створює змінні, значення яких не можна змінювати після ініціалізації. Як і у випадку з **let**, змінні, оголошені через **const**, мають блочну область видимості й повинні бути ініціалізовані одразу під час оголошення.

```
const PI = 3.14 // Оголошуємо константу PI та присвоюємо їй значення  
console.log(PI) // Виводимо значення PI
```

Оголошення **const** задає константу, тобто "змінну", яку не можна змінювати:

```
const apple = 5;  
apple = 10;    // помилка
```

В іншому оголошення **const** повністю аналогічно **let**.

Константа з об'єктом захищає лише посилання, але дозволяє змінювати властивості. *Те саме вірно, якщо константі присвоєно масив або інше об'єктне значення.*

```
const user = {  
  name: 'Вася'  
}
```

```
user.name = 'Петя' // дозволено
```

```
user = 5 // не можна, буде помилка
```

Коли використовувати camelCase та SNAKE_CASE для констант в JavaScript

У JavaScript, вибір між **camelCase** та **SNAKE_CASE** (**UPPER_SNAKE_CASE** або **SCREAMING_SNAKE_CASE**) для констант залежить від контексту х 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.

```
// 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
```

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**
- **Локальна область видимості**
 - **Функціональна область видимості** (для **var**)
 - **Блокова область видимості** (наприклад, у циклах або умовних операторах **{...}**)

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.

Best practices

Використовуйте виразні, зрозумілі назви змінних



```
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'
```



```
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
```

Не використовуйте var



`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



`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



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



```
const MILLISECONDS_IN_A_DAY = 86400000;  
setTimeout(tick, MILLISECONDS_IN_A_DAY); // Чітко і зрозуміло,
```

Явність краща за неявність



```
const locations = ['Austin', 'New York', 'San Francisco'];
locations.forEach((l) => {
  doStuff();
  doSomeOtherStuff ();

  // ...

  // ...

  // Стійте. Ще раз, що таке 'l'?
  dispatch(l);
});
```

Явність краща за неявність



```
const locations = ['Austin', 'New York', 'San Francisco'];
locations.forEach((location) => {
  doStuff();
  doSomeOtherStuff ();
  // ...
  // ...
  // ...
  dispatch(location);
});
```


Не добавляйте зайвий контекст



```
const car = {  
  carMake: 'Honda',  
  carModel: 'Accord',  
  carColor: 'Blue',  
};  
  
function paintCar(car) {  
  car.carColor = 'Red';  
}
```

Не додавайте зайвий контекст



```
const car = {  
  make: 'Honda',  
  model: 'Accord',  
  color: 'Blue',  
};
```

```
function paintCar(car) {  
  car.color = 'Red';  
}
```

Використовуйте аргументи за замовчуванням



```
function createMicrobrewery (name) {  
  const breweryName = name || 'Hipster Brew Co.';  
  // ...  
}
```



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

Аргументи функції , не більше трьох



```
function createMenu(title, body, buttonText, cancellable) {  
    // ...  
}
```

```
createMenu('Foo', 'Bar', 'Baz', true)
```

Аргументи функції , не більше трьох



```
function createMenu({ title, body, buttonText, cancellable }) {  
  // ...  
}
```

```
createMenu({  
  title: 'Foo',  
  body: 'Bar',  
  buttonText: 'Baz',  
  cancellable: true  
})
```

Функції повинні виконувати одну дію



```
function emailClients(clients) {  
  clients.forEach((client) => {  
    const clientRecord = database.lookup(client);  
    if (clientRecord.isActive()) {  
      email(client);  
    }  
  });  
}
```

Функції повинні виконувати одну дію



```
function emailClients(clients) {  
  clients.filter(isClientActive).forEach(email);  
}
```

```
function isActive(client) {  
  const clientRecord = database.lookup(client);  
  return clientRecord.isActive();  
}
```

Назви функцій мають відображати х ді



```
function addToDate(date, month) {  
  // ...  
}
```

```
const date = new Date();
```

// Из имени функции сложно понять, что она добавляет

```
addToDate(date, 1);
```


Назви функцій мають відображати х ді



```
function addMonthToDate (month, date) {  
  // ...  
}
```

```
const date = new Date();  
addMonthToDate(1, date);
```

Функції повинні мати один рівень абстракції



```
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
  })
}
```

Функції повинні мати один рівень абстракції



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

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

```
function parseBetterJSAlternative (code) {  
  const tokens = tokenize(code); // Виклик функції токенизації  
  const ast = buildAST(tokens); // Виклик функції побудови AST  
  // Розбиття коду на речення  
}
```

Уникайте дублювання коду



```
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);  });}
```

Уникайте дублювання коду



```
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);
});
}
```

Не використовуйте флаги як параметри функції



```
function createFile(name, temp) {  
  if (temp) {  
    fs.create(`./temp/${name}`);  
  } else {  
    fs.create(name);  
  }  
}
```

Не використовуйте флаги як параметри функці



```
function createFile(name) {  
  fs.create(name);  
}
```

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

Уникайте побічних ефектів



```
let name = 'John Smith';
```

```
function splitIntoFirstAndLastName () {  
    name = name.split(' ');  
}
```

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


Уникайте побічних ефектів



```
function splitIntoFirstAndLastName (name) {  
  return name.split(' ');  
}
```

```
const name = 'John Smith';  
const newName = splitIntoFirstAndLastName (name);  
console.log(name); // 'John Smith'  
console.log(newName); // ['John', 'Smith']
```

Перевага функціонального програмування



```
const programmerOutput = [  
  { name: 'John Smith', linesOfCode: 500 },  
  // ...  
];
```

```
let totalOutput = 0;
```

```
for (let i = 0; i < programmerOutput.length; i++) {  
  totalOutput += programmerOutput[i].linesOfCode;  
}
```

Перевага функціонального програмування



```
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);
```

Уникайте негативних умов



```
function isDOMNodeNotPresent(node) {  
    // ...  
}
```

```
if (!isDOMNodeNotPresent(node)) {  
    // ...  
}
```

Уникайте негативних умов



```
function isDOMnodePresent (node) {  
  // ...  
}
```

```
if (isDOMnodePresent (node)) {  
  // ...  
}
```

Видаляйте невикористаний код



```
function oldRequestModule (url) {  
  // ...  
}
```

```
function newRequestModule (url) {  
  // ...  
}
```

```
const req = newRequestModule;  
inventoryTracker ('android', req, 'www.inventory-awesome.io');
```

Видаляйте невикористаний код



```
function newRequestModule(url) {  
  // ...  
}
```

```
const req = newRequestModule;  
inventoryTracker('android', req, 'www.inventory-awesome.io');
```

Не залишайте закомментований код



```
doStuff():  
// doOtherStuff():  
// doSomeMoreStuff():  
// doSoMuchStuff();
```



```
doStuff();
```


Уникайте маркерів візуального розділення коду



```
////////////////////////////////////  
// Створення об'єкта
```

```
////////////////////////////////////  
$scope.model = {  
  menu: 'foo',  
  nav: 'bar',  
};
```

```
////////////////////////////////////  
// Встановлення змінної
```

```
////////////////////////////////////  
const actions = function () {  
  // ...  
};
```

Уникайте маркерів візуального розділення коду



```
$scope.model = {  
  menu: 'foo'.  
  nav: 'bar'.  
};
```

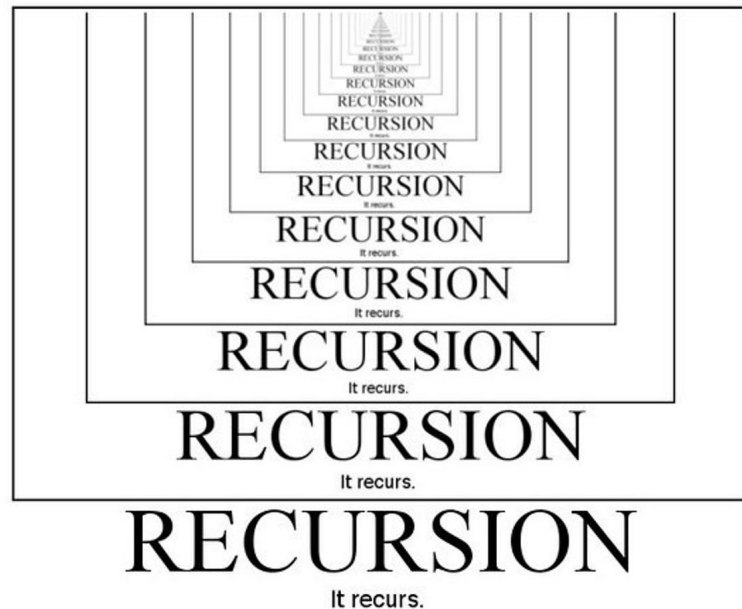
```
const actions = function () {  
  // ...  
};
```

Рекурсія

Рекурсія - це процес, в якому функція викликає саму себе як частину свого виконання.

Рекурсія у програмуванні **використовується**, коли задачу можна розбити на менші, схожі задачі.

Вона **дозволяє функції викликати саму себе** однією або декількома своїми версіями, доки не буде досягнута базова умова, яка завершує рекурсію.



Завдання 1: зведення числа x натуральний ступінь n . У цьому прикладі функція з параметрами (2,3).

```
function pow (x,n) {  
  if (n !== 1) { return x *= pow(x,n - 1); }  
  else { return x; }  
}
```

```
console.log( pow(2,3));    // 8
```

[Приклад...](#)

1. Викликаємо функцію параметрами (2,3), тобто. функція має цей вигляд:

```
function pow (2, 3) {  
  if (3 !== 1) {  
    return 2 *= pow(2, 3 - 1);  
  } else  
  { return  
    2;  
  }  
}
```

2. Далі функція (назвемо зовнішня) викликає сама себе та створює новий рівень вкладеності (назвемо "клон" 1). Зовнішня функція не може завершитись доки не завершаться всі внутрішні копії (клони).
3. Далі клон 1 викликає саму себе і створює новий рівень вкладеності (клон 2).
4. Далі на клоні 2 спрацьовує умова **else** → **return x** і клон 2 повертає двійку клону 1.
5. На клоні 1 спрацьовує умова **if** тобто **2*2** (двійку повернув клон 2) і результат (четвірка) повертається зовнішньо функції .
6. На зовнішню функцію спрацьовує умова **if** тобто. **2*4** (четвірку повернув клон 1), і повертає результат (вісімку) у **console.log()**.

alert(pow(2, 3))



```
function pow (2, 3) {  
  if (3 !== 1) {  
    return x = 2 * pow(2, 3 - 1);  
  } else {  
    return 2;  
  }  
}
```


alert(pow(2, 3))



```
function pow (2, 3) {  
  if (3 !== 1) {  
    return x = 2 * pow(2, 3 - 1);  
  } else {  
    return 2;  
  }  
}
```

3 - 1



```
function pow (2, 2) {  
  if (2 !== 1) {  
    return x = 2 * pow(2, 2 - 1);  
  } else {  
    return 2;  
  }  
}
```

alert(pow(2, 3))

function pow (2, 3) {
 if (3 !== 1) {
 return x = 2 * pow(2, 3 - 1);
 } **else** {
 return 2;
 }
}

3 - 1

function pow (2, 2) {
 if (2 !== 1) {
 return x = 2 * pow(2, 2 - 1);
 } **else** {
 return 2;
 }
}

2 - 1

function pow (2, 1) {
 if (1 !== 1) {
 return x = 2 * pow(2, 1 - 1);
 } **else** {
 return 2;
 }
}

alert(pow(2, 3))

function pow (2, 3) {
 if (3 !== 1) {
 return x = 2 * pow(2, 3 - 1);
 } **else** {
 return 2;
 }
}

3 - 1

function pow (2, 2) {
 if (2 !== 1) {
 return x = 2 * pow(2, 2 - 1);
 } **else** {
 return 2;
 }
}

2 - 1

function pow (2, 1) {
 if (1 !== 1) {
 return x = 2 * pow(2, 1 - 1);
 } **else** {
 return 2;
 }
}

Спрацьовує
умова **else**,
повертаючи
значення 2

alert(pow(2, 3))

function pow (2, 3) {
 if (3 !== 1) {
 return x = 2 * pow(2, 3 - 1);
 } **else** {
 return 2;
 }
}

3 - 1

function pow (2, 2) {
 if (2 !== 1) {
 return x = 2 * pow(2, 2 - 1);
 } **else** {
 return 2;
 }
}

2 - 1

function pow (2, 1) {
 if (1 !== 1) {
 return x = 2 * pow(2, 1 - 1);
 } **else** {
 return 2;
 }
}

2 * 2

function pow (2, 2) {
 if (2 !== 1) {
 return x = 2 * 2;
 } **else** {
 return 2;
 }
}

2

Спрацьовує
умова else,
повертаючи
значення 2

alert(pow(2, 3))

```
function pow (2, 3) {  
  if (3 !== 1) {  
    return x = 2 * pow(2, 3 - 1);  
  } else {  
    return 2;  
  }  
}
```

3 - 1

```
function pow (2, 2) {  
  if (2 !== 1) {  
    return x = 2 * pow(2, 2 - 1);  
  } else {  
    return 2;  
  }  
}
```

2 - 1

```
function pow (2, 1) {  
  if (1 !== 1) {  
    return x = 2 * pow(2, 1 - 1);  
  } else {  
    return 2;  
  }  
}
```

2 * 4

```
function pow (2, 3) {  
  if (3 !== 1) {  
    return x = 2 * 4;  
  } else {  
    return 2;  
  }  
}
```

2 * 2

```
function pow (2, 2) {  
  if (2 !== 1) {  
    return x = 2 * 2;  
  } else {  
    return 2;  
  }  
}
```

2

4

Спрацьовує
умова else,
повертаючи
значення 2

alert(pow(2, 3))

```
function pow (2, 3) {  
  if (3 !== 1) {  
    return x = 2 * pow(2, 3 - 1);  
  } else {  
    return 2;  
  }  
}
```

3 - 1

```
function pow (2, 2) {  
  if (2 !== 1) {  
    return x = 2 * pow(2, 2 - 1);  
  } else {  
    return 2;  
  }  
}
```

2 - 1

```
function pow (2, 1) {  
  if (1 !== 1) {  
    return x = 2 * pow(2, 1 - 1);  
  } else {  
    return 2;  
  }  
}
```

Спрацьовує
умова else,
повертаючи
значення 2

2

2 * 2

```
function pow (2, 2) {  
  if (2 !== 1) {  
    return x = 2 * 2;  
  } else {  
    return 2;  
  }  
}
```

4

2 * 4

```
function pow (2, 3) {  
  if (3 !== 1) {  
    return x = 2 * 4;  
  } else {  
    return 2;  
  }  
}
```

alert 8

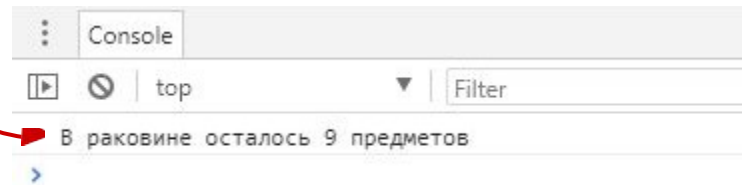


Завдання 2

Реалізувати алгоритм миття посуду:

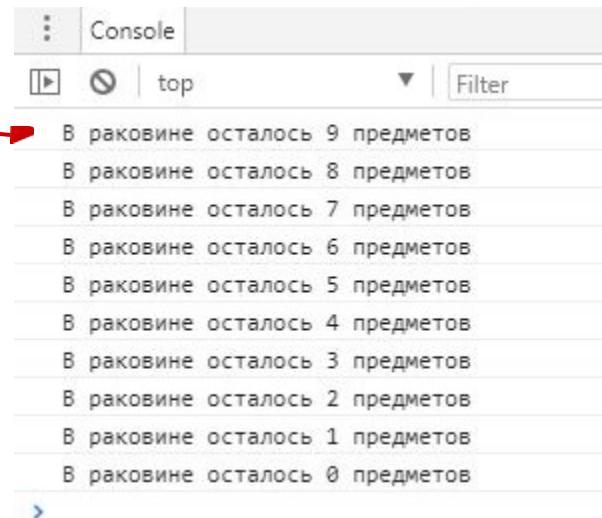
1. Взяти будь-який предмет з раковини
2. Помити його і відкласти убік
3. Якщо у раковині ще щось є – повторити

```
let washNextItem = function (itemsLeft)
  { itemsLeft --;
    console.log('У раковині залишилося ' + itemsLeft + ' предметів');
  };
washNextItem(10);
```



Робимо з функції
рекурсію...

```
let washNextItem = function (itemsLeft)
  { itemsLeft --;
  console.log('У раковині залишилося ' + itemsLeft + ' предметів');
  if (itemsLeft > 0) {
    washNextItem(itemsLeft);
  }
};
washNextItem(10);
```



Той самий приклад можна переписати за допомогою циклу **while...**

```
let washNextItem = function (itemsLeft)
  { while (itemsLeft--) {
    console.log('У раковині залишилося ' + itemsLeft + ' предметів');
  }
};
washNextItem(10);
```

