

Лекція 10

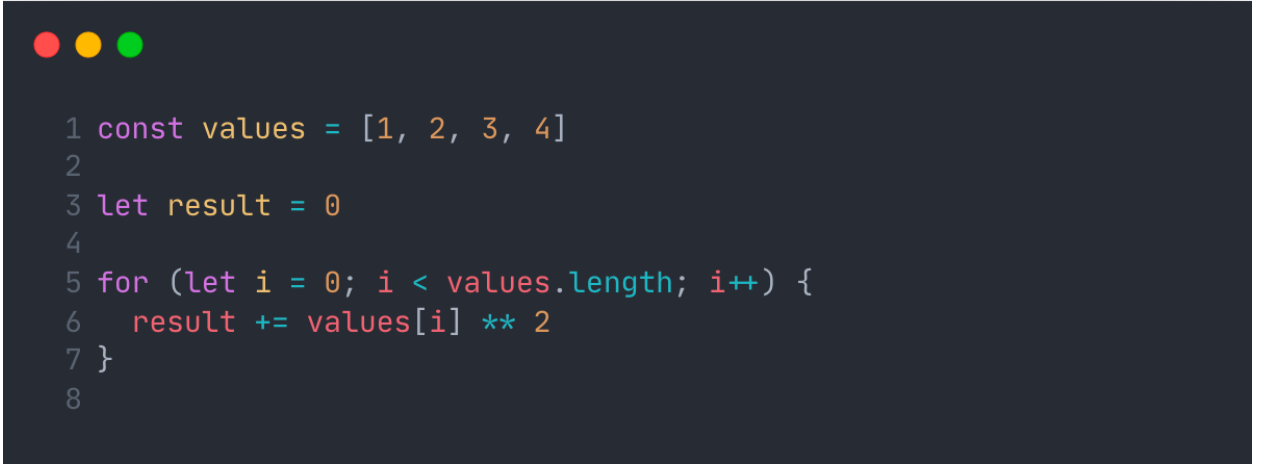
ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ

10.1. Імперативний і декларативний підходи у програмуванні

Функціональне програмування – це підхід у програмуванні, який спирається на обчислення виразів (декларативний підхід), а не на послідовне виконання команд (імперативний підхід). Функціональне програмування відрізняється від об'єктно-орієнтованого, тому деякі методики, які успішно працювали в першому, мало пристосовані до другого і навпаки.

Що це означає на практиці?

Розглянемо приклади, які демонструють відмінність імперативного підходу від декларативного у написанні коду:



```
1 const values = [1, 2, 3, 4]
2
3 let result = 0
4
5 for (let i = 0; i < values.length; i++) {
6   result += values[i] ** 2
7 }
8
```

Приклад 1: Імперативний підхід

Припустимо, існує масив чисел. Наше завдання – отримати суму його елементів, зведену у квадрат. Чому цей код можна вважати імперативним? По-перше – це цикли, набір дій, послідовне виконання яких веде до вирішення задачі. Цикли дуже характерні для імперативного підходу у програмуванні. Ще одна відмінність – це мутації, які передбачають зміни вхідних аргументів у коді. Наведений фрагмент коду орієнтується на чітке та послідовне виконання команд для отримання результату.

Розглянемо приклад декларативного підходу написання коду:

```
1 const values = [1, 2, 3, 4]
2
3 const getSumSquare = (arr) => arr.reduce((acc, value) => acc +
  value ** 2, 0)
4
5 const result = getSumSquare(values)
6
```

Приклад 2: Декларативний підхід

Задача та сама, але тепер ми не використовуємо цикли та мутації. Натомість ми отримуємо результат за допомогою функції *reduce*. У декларативному підході ми орієнтуємося на кінцевий результат, оминаючи прописування конкретних кроків, необхідних для його досягнення.

10.2 Чисті функції

Чисті функції – це основні блоки функціонального програмування.

Концепція чистих функцій складається з наступних пунктів:

Перший: Same input = Same output. Це означає, що за одних і тих самих входніх параметрів функція завжди буде повертати той самий результат. Саме тому чисті функції передбачувані та стабільні.

Другий: у чистих функцій немає побічних ефектів. У чистої функції має бути лише одна задача (одна роль або відповідальність). Якщо ваша функція повертає суму чисел, вона повинна робити тільки це і не відповідати за виконання сторонніх задач.

Третій: чисті функції не залежать від зовнішніх даних. У цих функціях ми ніколи не використовуємо глобальних змінних.

Наведемо приклади, щоб зрозуміти відмінність чистих функцій від інших.

```
1 const formatDate = () => new Date().toLocaleDateString()
2
```

Приклад 3: Функція, що не є чистою функцією

Ця функція не є чистою функцією, оскільки результат виконання залежить від зовнішніх даних. Натомість викликається нова дата `Date()`, після чого відбувається форматування. Ця функція завжди буде видавати різний результат залежно від дати в момент виконання. Її результат відрізнятиметься сьогодні, завтра та післязавтра.

Для порівняння подивимося, як виглядає код чистої функції для цього завдання:

```
1 const formatDate = (date) => date.toLocaleDateString()  
2
```

Приклад 4: Чиста функція

Для того, щоб перетворити функцію з прикладу 3 в чисту, нам лише потрібно передавати дату (`date`) як вхідний аргумент. Тепер ми впевнені: якщо передаємо певну дату, нам повернеться певний результат.

Розглянемо ще один приклад:

```
1 let initialValue = 1  
2  
3 const add = (a) => {  
4   initialValue += a  
5   return initialValue  
6 }  
7  
8 const result1 = add(1) // 2  
9 const result2 = add(1) // 3  
10 const result3 = add(1) // 4  
11
```

Приклад 5: Код функції, яка не є чистою функцією

У нас є глобальне значення `initialValue`, оголошене в рядку 1. Крім цього, у нас є функція `add`, яка приймає за вхідний аргумент деяке число (`a`) і, як результат, здійснює підсумовування цього числа з нашою глобальною змінною (5 рядок коду). Після цього в рядках коду 8, 9 і 10 ми викликаємо цю функцію тричі, але щоразу вона видає нам різний результат. Це відбувається

тому, що використовується глобальна змінна *initialValue*, внаслідок чого функція викликана з тими самими аргументами повертає різний результат.

Перетворимо цей код на код чистої функції:

```
1 const add = (a, b) => a + b
2
3 const result1 = add(1, 2) // 3
4 const result2 = add(1, 2) // 3
5 const result3 = add(1, 2) // 3
6
```

Приклад 6: Чиста функція

У цьому коді ми спираємося лише на вхідні аргументи. Ми приймаємо два з них (*a*, *b*), а потім повертаємо їхню суму. Такий код застрахований від впливу зовнішніх змінних і завжди буде стабільним і передбачуваним.

Звичайна редукція

Для продовження розгляду чистих функцій та їх композицій розглянемо функцію редукції.

Функція *reduce()* в JavaScript один з основних з точки зору функціонального програмування методів.

Припустимо потрібно підрахувати суму елементів завданого масива.

Це можна зробити за допомогою наступного коду:

```
const euros = [29.76, 41.85, 46.5];
const sum = euros.reduce((total, amount) => total + amount);
sum // 118.11
```

Розглянемо як працює наведений код:

- Функція *reduce()* отримує два параметра: *total* та число, з яким в даний момент працює функція.
- Метод перебирає всі числа в масиві, так ніби це є циклом *for*.
- На початку роботи циклу, змінна *total* має значення першого числа з початку масива (29.76), а числом в обробці є наступне в цьому ж масиві число (41.85).

- Для виконання цієї конкретної задачі необхідно додати число, яке зараз в обробці до змінної *total*.
- Таке обчислення повторюється для кожного числа з масиву і кожного разу число в обробці змінюється на наступне число в масиві, яке розташоване праворуч.

Коли чисел в масиві вже нема, метод повертає значення *total*.

У наведеному в прикладі коді використано синтаксис ES6. Цей фрагмент коду може бути поданий у наступному вигляді:

```
var euros = [29.76, 41.85, 46.5];
var sum = euros.reduce( function(total, amount){
    return total + amount
});sum // 118.11
```

За написання коду з використанням синтаксису ES6 замінюють *var* на *const* слово *function* на символ *=>* (стрілочна функція) після параметрів, та опускають ключове слово *return*.

Розглянемо приклад коду для підрахунку середньоарифметичного значення масиву з використанням функції *reduce()* в JavaScript.

В цьому випадку замість суми елементів масива слід повертати суму поділену на кількість елементів в масиві.

Це можна виконати за використання аргументів, які має функція *reduce()*, одним є *index*, який показує кількість повторень, що їх виконала функція. Останній аргумент функції *reduce()* в цьому прикладі це вже сам масив.

```
const euros = [29.76, 41.85, 46.5];
const average = euros.reduce((total, amount, index, array) => {
    total += amount;
    if( index === array.length-1) {
        return total/array.length;
    }else {
        return total;
    }
});average // 39.37
```

10.3 Композиція функцій

Розглянемо композицію функцій. Під час роботи наJavaScript, ви, напевно, стикалися з цим процесом, навіть не підозрюючи про це. Композиція функції є процесом передачі результату виконання однієї функції на вхід іншої функції.

Розглянемо на прикладі, що це значить:

```
1 const square = (x) => x ** 2
2 const addTwo = (x) => x + 2
3
4 const result = square(addTwo(4))
5
```

Приклад 7: Композиція функцій

Припустимо, у нас є дві функції: одна підносить змінну x в квадрат, а інша – додає до неї 2. Ми можемо викликати дві ці функції й отримати результат. Відповідно до законів математики, спочатку буде викликана функція *addTwo*, а потім – результат буде піднесений у квадрат за допомогою *square*. Можливо створити більш абстрактну функцію, яка зможе композувати дві або більше функцій:

```
1 const compose =
2   (...fns) =>
3   (x) =>
4     [...fns].reverse().reduce((acc, fn) => fn(acc), x)
5
6 const repeatTwice = (str) => `${str} ${str}`
7 const addExclamation = (str) => `${str}!`
8
9 const withCompose = compose(repeatTwice, addExclamation)
10
11 withCompose('hello') // hello! hello!
12
```

Приклад 8: Compose

Функція *compose* бере на себе N-кількість інших функцій. Потім функція повертатиме функцію, до якої був застосований вхідний аргумент, і так до всіх

функцій по черзі. Зауважимо, що в 11 рядку коду (де видно виклик кінцевого результату) дії з виклику функцій виконуються справа-наліво: спочатку виконується дія з додаванням до слова *hello* знака оклику, а потім дублюється слово, в результаті чого на екран виводиться: *hello! hello!*

Якщо ви бажаєте інвертувати цей процес виклику функцій, можете скористатися функцією *pipe*:

```
1 const pipe =  
2   (...fns) =>  
3   (x) =>  
4     [...fns].reduce((acc, fn) => fn(acc), x)  
5  
6 const repeatTwice = (str) => `${str} ${str}`  
7 const addExclamation = (str) => `${str}!`  
8  
9 const withPipe = pipe(repeatTwice, addExclamation)  
10  
11 withPipe('hello') // hello hello!  
12
```

Приклад 9: Pipe

Ця функція виконує аналогічне завдання, не перевертаючи масив функцій і звично виконуючи його дії в послідовності зліва-направо: спочатку застосовується функція *repeatTwice* (повторення) і лише потім *addExclamation* (додавання знака оклику).

Імутабельність

Ще одна важлива концепція функціонального програмування – це імутабельність. Вона передбачає відсутність мутацій і роботу з константами. Створюючи один раз будь-яку змінну, ви втрачаєте можливість змінити її в майбутньому. Це робить код більш передбачуваним. У результаті вдається уникнути появи раніше не прогнозованих багів.

Щоб уникнути непередбачуваної поведінки в коді, слід не мутувати (не змінювати) дані, а перезаписувати їх. Це стосується як масивів, так і об'єктів.

```
1 const simpleConstant = 1
2 simpleConstant = 2 // Error
3
4 const objectConstant = {
5   a: 1,
6   b: 2,
7 }
8 objectConstant.a = 2 // No error
9
```

Приклад 10: Імутабельність

Якщо ми створимо константу JavaScript (1 рядок), а потім спробуємо її перезаписати (2), то з'являється помилка. Однак, якщо створити константу об'єкта (4) і спробувати перезаписати поле об'єкта (8), ми не отримаємо помилку.

Важливо розуміти, що повної імутабельності досягти досить складно й іноді зміни все ж таки будуть відбуватися. Однак за допомогою таких готових рішень як *Immutable* та *Immer*, цей процес можна звести до мінімуму.

Метод *Immutable.js*, який надає можливості роботи з великою кількістю структур даних, що мають властивість постійної імутабельності (Persistent Immutable), включає: *List*, *Stack*, *Map*, *OrderedMap*, *Set*, *OrderedSet* та *Record*.

Immer (від німецького «завжди») – невеличкий пакет, який спрощує роботу з незмінними станами даних.

10.4 Рекурсивна функція і функції вищого порядку

Рекурсивна функція – це функція, що викликає сама себе. Рекурсивна функція може розглядатися як заміна циклам у функціональному програмуванні. Слід пам'ятати, що рекурсія завжди повинна мати умови для виходу. В іншому випадку за досягнення максимальної глибини рекурсії заповниться *Call Stack*, що загрожує зациклюванням та відмові роботи програми.

Розглянемо приклад коду рекурсивної функції:


```
1 const factorial = (n) => (n >= 1 ? n * factorial(n - 1) : 1)
2
```

Приклад 11: Рекурсивна функція

Перед вами проста рекурсивна функція, призначена для обчислення факторіалу. Тут можна побачити, як відбувається перевірка вхідного аргументу. Якщо він не дорівнює 1, функція повертає добуток параметра n на результат виклику самої себе від попереднього значення параметра (тобто факторіал $(n-1)$), інакше – повертає 1 (тобто факторіал 1).

Різновидом функцій вищого порядку (*Higher-order Functions* або *HOF*) також є комбіновані функції, які можуть приймати в себе функції у вигляді вхідного аргументу, повертати їх як результат або робити те й інше одночасно. До цих функцій належать *map*, *reduce*, *filter*.

Map – це колекція ключ/значення (подібно до *object*). *Map* дозволяє використовувати ключі будь-якого типу.

З *map* використовують наступні методи, має наступні властивості:

new Map() – створює колекцію.

map.set(key, value) – записує за ключем *key* значення *value*.

map.get(key) – повертає значення за ключем або *undefined*, якщо ключ *key* відсутній.

map.has(key) – повертає *true*, якщо ключ *key* присутній в колекції, інакше повертає *false*.

map.delete(key) – видаляє елемент за ключем *key*.

map.clear() – очищує колекцію від всіх елементів.

map.size – повертає поточну кількість елементів в масиві.

```
const array1 = [1, 4, 9, 16];
```

Приклад використання *map()*.

```
// pass a function to map
```

```
const map1 = array1.map(x => x * 2);
```

```
console.log(map1);
```

```
// expected output: Array [2, 8, 18, 32]
```

10.5 Карування

Карування – це підхід до написання функції, в якому вона завжди буде приймати лише один аргумент. Ця методика дозволяє викликати функцію частково.

Порівняємо JavaScript код функцій з каруванням та без нього:

```
1 const multiply = (a, b, c) => a * b * c
2 const result = multiply(2, 3, 4)
3
```

Приклад 12: Функція без карування

У першому рядку коду зображено звичайну функцію *multiply*. Вона отримує три аргументи, після чого обраховує добуток цих аргументів. Перепишемо цей код таким чином, щоб отримати функцію з каруванням:

```
1 const multiply = (a) => (b) => (c) => a * b * c
2 const result = multiply(2)(3)(4)
3
```

Приклад 13: Функція з каруванням

Ми будемо повертати функцію по черзі до тих пір, поки вона не поверне всі наші вхідні аргументи. Завдяки цій дії ми матимемо можливість виклику функції лише частково.

Розглянемо використання часткового виклику за карування на наступному прикладі.

Є функція виведення повідомлення *log(date, importance, message)*, яка форматує та виводить інформацію для користувача:

```
function log(date, importance, message) {
  alert(`[${date.getHours()}]:${date.getMinutes()}]
[${importance}] ${message}`);
}
```

Застосуємо до неї карування:

```
log = _.curry(log);
```

В даному випадку, за допомогою наступної додаткової функції *curry()*:

```
function curry(f) { // curry(f) виконує карування
  return function(a) {
    return function(b) {
      return function(c) {
        return f(a, b, c);
      };
    };
  };
}
```

Після цього функція *log()* може продовжувати працювати і так як до карування:

```
log(new Date(), "DEBUG", "some debug"); // log(a, b, c)
```

А може працювати з урахуванням карування:

```
log(new Date())("DEBUG")("some debug"); // log(a)(b)(c)
```

Але після карування ми можемо викликати цю функцію частково, так би мовити, для окремих її параметрів окремо:

```
// logNow буде частковим застосуванням функції log з
фіксованим першим аргументом
let logNow = log(new Date());
```

```
// використаємо функцію logNow()
logNow("INFO", "message"); // виведення [HH:mm] INFO message
```

Тепер *logNow()* – це функція *log()* з фіксованим першим аргументом, іншими словами, «частково застосована» або «часткова» функція.

Таким чином:

1. Функція *log()* нічого не втратила внаслідок карування: її можна застосовувати так само як і до карування.
2. Після карування можна утворювати частинні функції на підставі функції *log()*.

10.6 Особливості нефункціонального підходу у програмуванні

З практичної точки зору головною відмінністю функціонального підходу у програмуванні є те, що код не має містити наступних елементів:

1 Циклів (*while*, *do...while*, *for*, *for...of*, *for...in*). Цикли вважаються імперативними, тому при використанні функціонального підходу необхідно змінювати їх на рекурсію.

2 Оголошень *let* або *var*. Ці змінні забезпечують мутації, а у функціональному програмуванні в першу чергу керуються принципом імутабельності.

3 Функцій *void()*. Вони є побічними ефектами, тому у функціональному програмуванні від них доводиться відмовитися.

4 Мутації об'єктів. Замість зміни об'єкта необхідно створювати на його основі новий (перезаписати).

5 Мутуючих методів для масивів (*copyWithin*, *fill*, *pop*, *push*, *reverse*, *shift*, *unshift*, *sort*, *splice*). Всі ці методи призводять до мутації масиву вхідних аргументів, тому їм краще знайти заміну серед *map*, *reduce* та інших.

Бібліотеки Ramda та Sanctuary

Ramda.js –бібліотека, яка спрощує написання коду у функціональному стилі, включає в себе різні функції і є найзручнішою для початку ознайомлення з функціональним програмуванням на JS. Ramda.js містить багато функцій-помічників (хелперів), які істотно полегшують написання коду. Ця бібліотека не містить алгебраїчні специфікації даних, лише набір карірованих функцій.

Розглянемо роботу в Ramda на прикладі:

```
1 const R = require('ramda')
2
3 const classyGreeting = (firstName, lastName) =>
4   `The name's ${lastName}, ${firstName} ${lastName}`
5
6 const yellGreeting = R.compose(R.toUpper, classyGreeting) //
   THE NAME'S BOND, JAMES BOND
7
8 R.compose(Math.abs, R.add(1), R.multiply(2))(-4) // 7
9
```

Приклад 15: R.Compose

На 3 та 4 рядках коду можна побачити функцію `classyGreeting`, яка конструює вітання залежно від імені та прізвища. На 6 рядку коду зображено виклик функції `R.compose`, яка спочатку викликає функцію `classyGreeting`, а потім використовує функцію-хелпер `R.toUpper`, яка дозволяє внести результат у upper case. На 8 рядку можна побачити вже інший, складніший приклад використання `R.compose` з каруванням.

```
1 const greet = R.replace('{name}', R._, 'Hello, {name}!')
2 greet('Alice') // 'Hello, Alice!'
3
```

Приклад 16: R._

Бібліотека `Ramda` дозволяє навіть пропускати аргументи, використовуючи замість них `R`. Безліч інших реальних прикладів використання `Ramda` можна знайти в Cookbook за [посиланням](#).

`Sanctuary` – це бібліотека, яка реалізує специфікації `Fantasy Land` (на відміну від `Ramda`). У ній представлені всі необхідні алгебраїчні структури та специфікації, що трохи ускладнює поріг входження при використанні бібліотеки.

Розглянемо, як реалізується задача щодо повернення кількості слів на прикладі `Sanctuary`:

```
1 const S = require('sanctuary')
2
3 const sample =
4   'Lorem Ipsum is simply dummy text of the printing and
5     typesetting industry.'
6 const wordsCount = S.pipe([S.words, S.size])
7
8 wordsCount(sample) // 12
9
```

Приклад 17: Повернення кількості слів у тексті Sanctuary

На 6 рядку коду можна побачити застосування функції `S.pipe`, яка містить 2 helper-функції. Вони відповідають за підрахунок слів у тексті.

10.7 Основні переваги і недоліки функціонального програмування

Переваги:

1 Код, що читається: математична логіка коду, написаного в декларативному стилі, набагато простіша для сприйняття.

2 Спрощена процедура тестування. Для тестування коду у ФП не потрібно створювати додаткових умов: слід просто передати вхідні аргументи, а потім перевірити отриманий результат. Внаслідок цього процес тестування займає набагато менше часу.

3 Спрощене налагодження: чисті функції набагато простіше дебатувати.

4 Передбачуваний код: методи функціонального програмування гарантують передбачувану поведінку коду, що зводить до мінімуму можливість раптової появи помилок.

Недоліки:

Високий поріг входження: функціональне програмування досить велика тема, для глибокого занурення на яку потрібен час, що часто і відлякує багатьох програмістів-початківців.

Менш продуктивний код. JavaScript за замовчуванням не є повністю функціональною мовою програмування. Тому перед тим, як писати код у стилі ФП, необхідно зрозуміти, чи дійсно це є необхідним. Частіше найкращим варіантом є комбінування методів функціонального та об'єктно-орієнтованого програмування.

Матеріал за посиланням

<https://dou.ua/forums/topic/37548/>

Посилання для детальнішого ознайомлення з функціональним програмуванням:

- <https://ramdajs.com/> Документація Ramda
- <https://github.com/sanctuary-js/sanctuary> Документація Sanctuary
- <https://github.com/fantasyland/fantasy-land> Документація Fantasy Land.