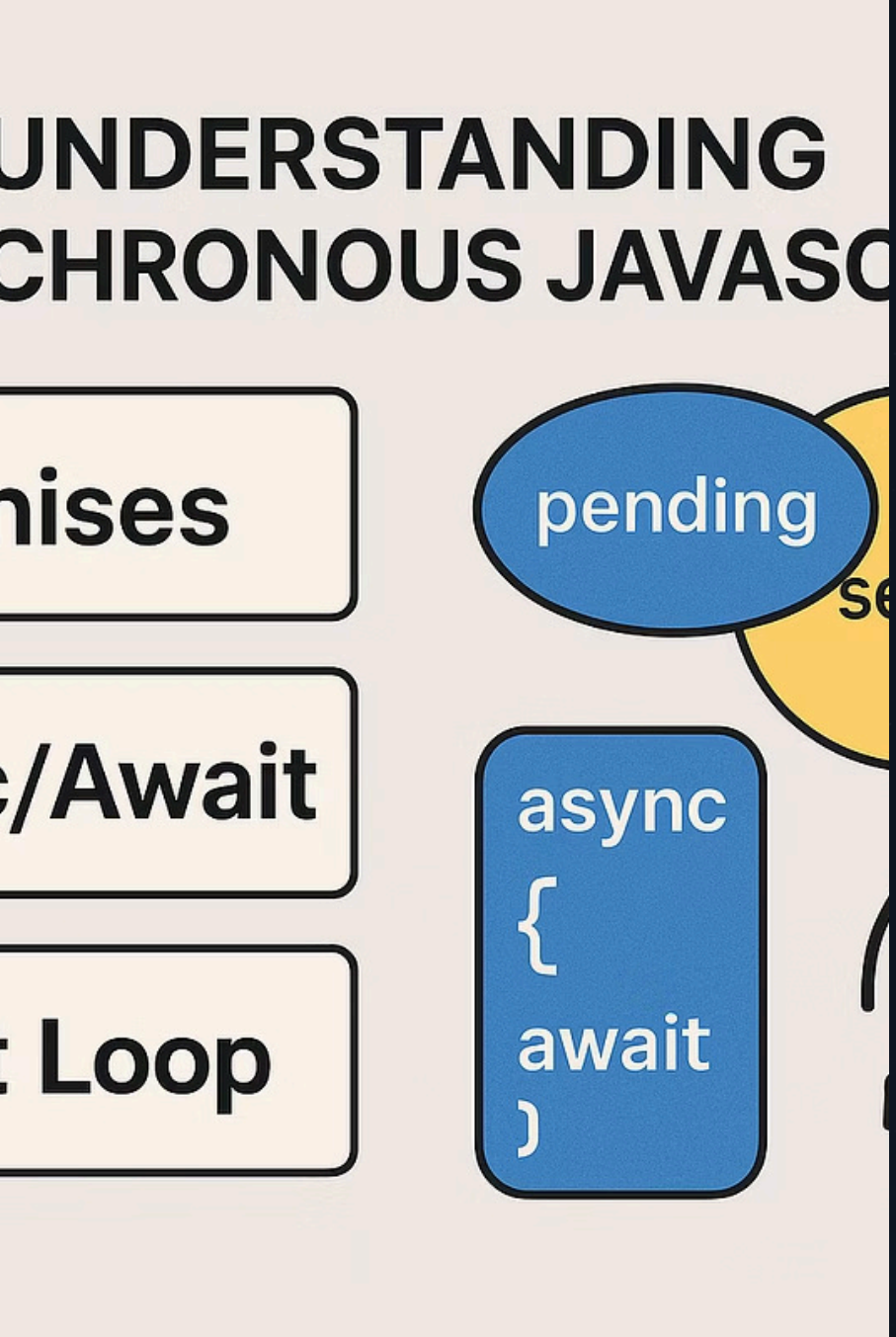


Promises & fetch API

Глибоке занурення у світ асинхронного JavaScript: від callbacks до сучасних промісів

Що таке асинхронний код?

Проблема синхронності

JavaScript є однопоточною мовою програмування, що означає один основний потік виконання коду. Але оточення JS (браузер чи Node.js) надає безліч API для роботи з подіями, запитами до серверу, файловою системою та базами даних.

Уявіть месенджер, де для відправки нового повідомлення потрібно чекати відповіді сервера, а весь застосунок при цьому заморожений. Неможливо навіть дізнатися про нові повідомлення без окремого запиту. Звучить жахливо, чи не так?

Але як це зробити з одним потоком? Відповідь — **асинхронний код**.

Краще рішення

Відправляти запити у фоні, продовжувати користування застосунком, а коли прийде відповідь — обробити її та показати зміни.

Переваги асинхронного програмування

Паралельне виконання

Код виконується поза основним потоком і не блокує його, працюючи паралельно з іншими операціями

Висока продуктивність

Тривалі операції виконуються у фоновому режимі, забезпечуючи швидкодію програми

Відмовостійкість

Програма залишається доступною для користувачів навіть під час виконання складних операцій

Синхронне програмування

- Послідовне виконання крок за кроком
- Блокування при тривалих операціях
- Програма стає недоступною

Асинхронне програмування

- Паралельне виконання завдань
- Без блокування головного потоку
- Постійна доступність застосунку

Функції зворотного виклику (Callbacks)

Callback functions — це функції, які передаються як аргументи іншим функціям для виконання після завершення певної операції або події.

01

Передача функції

Функція передається як параметр іншій функції

02

Виконання операції

Основна функція виконує свою роботу

03

Виклик callback

Після завершення викликається передана функція

Приклад: поїздка додому

Батьки їдуть додому на таксі (~15 хвилин). Замість постійних дзвінків кожні 5 хвилин, просимо їх передзвонити, коли будуть вдома.

```
function letsGoHome(cbFunction) {
  console.log('Call a taxi');
  console.log('On the road...');
  cbFunction();
}

function callback() {
  console.log('We are at home!');
}

letsGoHome(callback);
```

Важливо: Передаємо посилання на функцію, а не її виклик! `setTimeout(saySmth, 1000)` ✅ vs `setTimeout(saySmth(), 1000)` ❌

Таймери: setTimeout і setInterval

setTimeout

Виконує код **один раз** після затримки

```
setTimeout(callback, delay, ...args);

// Приклад
function saySmth(phrase, name) {
  console.log(`${phrase} ${name}!`);
}

setTimeout(saySmth, 1000, 'Hello', 'John');
```

Скасування: clearTimeout

```
const timerId = setTimeout(saySmth, 1000);
clearTimeout(timerId);
```

setInterval

Виконує код **періодично** через інтервал

```
setInterval(callback, delay, ...args);

// Приклад
setInterval(saySmth, 1000, 'Hello', 'John');
```

Скасування: clearInterval

```
const intervalId = setInterval(saySmth, 1000);
clearInterval(intervalId);
```

Рекурсивний setTimeout

Для ситуацій, коли потрібно дочекатися завершення функції перед плануванням наступного виклику:

```
let timerId = setTimeout(function tick() {
 // якийсь код
 timerId = setTimeout(tick, 2000);
}, 2000);
```

Event Loop: серце асинхронності

Event Loop — це механізм, що керує виконанням асинхронного коду в JavaScript. Він працює як у браузерах, так і в Node.js.

Stack (Стек)

Структура LIFO для виконання синхронного коду

Callback Queue

Черга FIFO для готових асинхронних коллбеків

Event Loop

Механізм, що циклічно перевіряє чергу і переносить коллбеки в стек

Як працює Event Loop?

1

API готує коллбеки

Асинхронні операції завершуються

2

Коллбеки в черзі

Готові функції переносяться в чергу

3

Перевірка Stack

Чи пустий зараз стек виконання?

4

Виконання

Коллбек з черги переноситься в Stack

Event Loop не блокує виконання: він періодично перевіряє чергу і виконує доступні завдання, не зупиняючи програму.

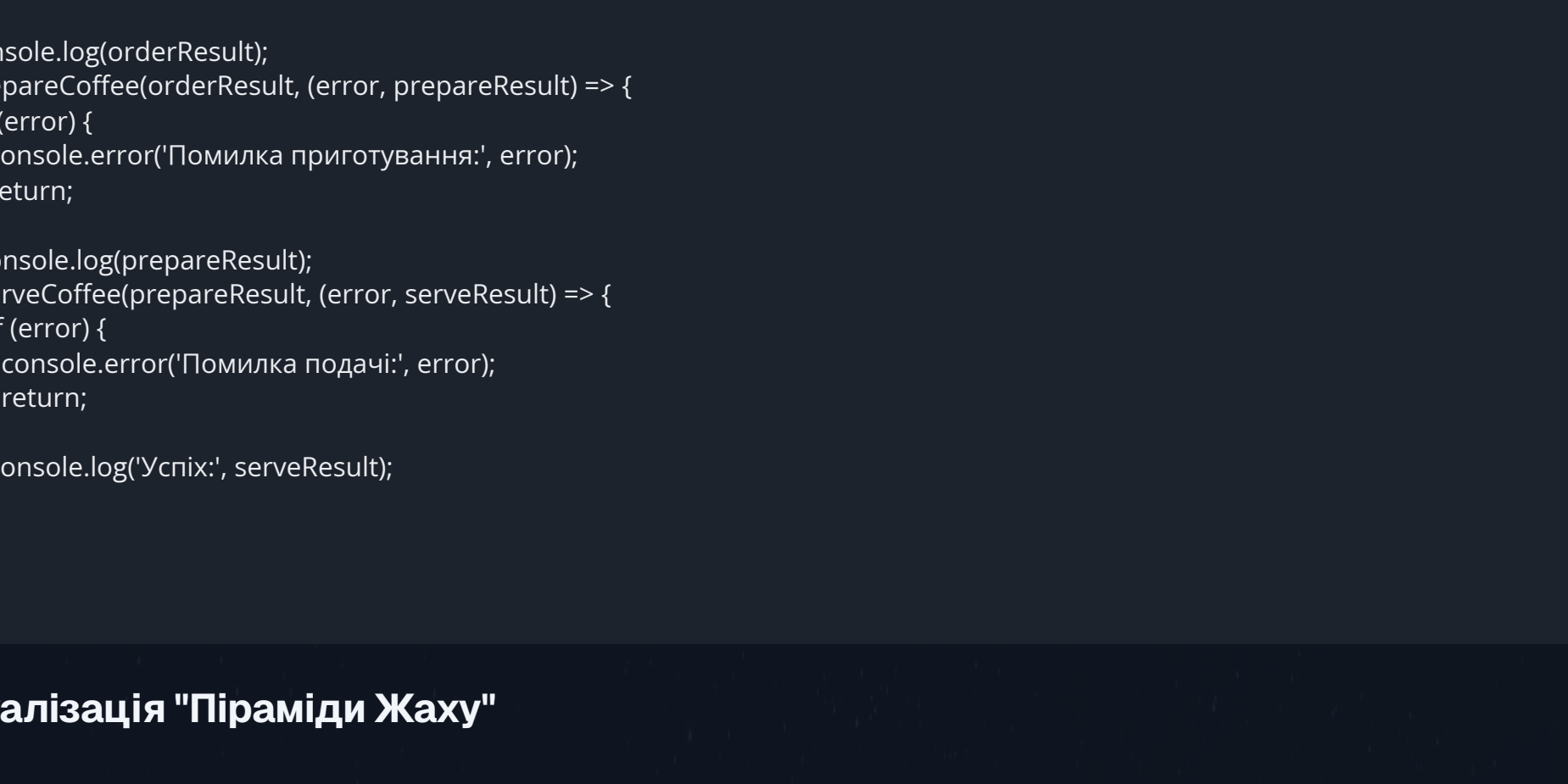
Callback Hell: проблема вкладеності

Коллбеки чудові для простих випадків, але при складній логіці виникає **callback hell** — надмірна вкладеність функцій, що робить код важким для читання та підтримки. Це часто називають "пірамідою жаху" (pyramid of doom).

Приклад: система замовлення в кафе

```
// Просте меню кафе
const menu = {
  americano: { price: 2.5, time: 1000 },
  latte: { price: 3.0, time: 1500 },
  cappuccino: { price: 3.2, time: 2000 }
};

// Функція замовлення напою
function placeOrder(coffeeType, callback) {
  console.log(`Замовляємо ${coffeeType}...`);
  setTimeout(() => {
    if (!menu[coffeeType]) {
      callback(`Кави "${coffeeType}" немає в меню.`, null);
      return;
    }
    callback(null, `${coffeeType} замовлено.`);
  }, 500); // Симулюємо затримку замовлення
}

// Функція приготування напою
function prepareCoffee(order, callback) {
  const coffeeType = order.split(' ')[0]; // Витягуємо тип кави з рядка замовлення
  console.log(`Готуємо ${coffeeType}...`);
  setTimeout(() => {
    callback(null, `${coffeeType} приготовлено.`);
  }, menu[coffeeType].time); // Симулюємо час приготування
}

// Функція подачі напою
function serveCoffee(preparedCoffee, callback) {
  console.log(`Подаємо ${preparedCoffee}...`);
  setTimeout(() => {
    callback(null, `Насолоджуйтесь ${preparedCoffee}!`);
  }, 300); // Симулюємо час подачі
}

// Приклад Callback Hell: Замовлення лате
placeOrder('latte', (error, orderResult) => {
  if (error) {
    console.error('Помилка замовлення:', error);
    return;
  }
  console.log(orderResult);
  prepareCoffee(orderResult, (error, prepareResult) => {
    if (error) {
      console.error('Помилка приготування:', error);
      return;
    }
    console.log(prepareResult);
    serveCoffee(prepareResult, (error, serveResult) => {
      if (error) {
        console.error('Помилка подачі:', error);
        return;
      }
      console.log('Успіх:', serveResult);
    });
  });
});
```

Візуалізація "Піраміди Жаху"



Ключові проблеми Callback Hell

Важко читати

Глибока вкладеність функцій ускладнює розуміння логіки коду та його візуальне сканування.

Складність налагодження

Виявлення та виправлення помилок стає значно складнішим через заплутаний потік виконання.

Проблеми з обробкою помилок

Кожна вкладена функція потребує окремої логіки обробки помилок, що веде до дублювання та ускладнення коду.