Лабораторна робота №3: Платформа Ethereum та мова програмування Solidity

Мета роботи: Вивчити архітектурні особливості мережі Ethereum, опанувати механіку газу та розробити свій перший смартконтракт на мові Solidity.

1. Теоретична частина

1.1. Архітектура Ethereum: Світовий комп'ютер

Ethereum — це не просто платіжна система, а децентралізована платформа для виконання коду. Її часто називають **"World Computer"** (Світовий комп'ютер), оскільки кожна нода в мережі виконує ті самі інструкції, досягаючи консенсусу щодо результату.

- **Віртуальна машина Ethereum (EVM):** Це рантайм-середовище для смартконтрактів. EVM є повною за Тюрінгом (з урахуванням обмежень на газ), що дозволяє реалізовувати будь-яку логіку.
- **Стан мережі (World State):** На відміну від Bitcoin (модель UTXO), Ethereum використовує **Account-based model**. Стан — це велика база даних, де зберігаються баланси адрес та стан (storage) усіх смартконтрактів.
- **Транзакції як перехід стану:** Будь-яка взаємодія (переказ коштів або виклик функції) — це транзакція. Математично це виглядає як функція переходу: , де — стан, а — транзакція.

1.2. Економіка газу (Gas)

Щоб запобігти нескінченним циклам (проблема зупинки Тюрінга) та спаму, виконання коду в Ethereum є платним.

- **Gas Limit:** Максимальна кількість "одиниць роботи", яку ви готові витратити на транзакцію.
- **Gas Price:** Ціна, яку ви платите за кожну одиницю газу (вимірюється в Gwei, де).
- **EIP-1559:** Сучасна модель оплати включає:
    - **Base Fee:** Базова плата мережі (спалюється).
    - **Priority Fee (Tip):** Чайові валідатору за пріоритетне включення в блок.

1.3. Основи Solidity

Solidity — це високорівнева об'єктно-орієнтована мова для написання смартконтрактів.

- **Типи даних:**
  - uint256 — беззнакове ціле число.
  - address — ідентифікатор гаманця (20 байт).
  - mapping(key => value) — хеш-таблиця (key-value storage).
  - struct — кастомний тип даних для групування змінних.
- **Модифікатори (modifier):** Використовуються для зміни поведінки функцій (наприклад, перевірка прав доступу).
- **Події (events):** Дозволяють зовнішнім застосункам (frontend) "підписуватися" на зміни в контракті. Дані подій зберігаються в логах блоку, а не в стані контракту (це дешевше).
- **Обробка помилок:**
  - require(умова, "повідомлення") — для перевірки вхідних даних (повертає залишок газу).
  - revert() — негайна зупинка з відкатом змін.
  - assert() — для критичних внутрішніх помилок (спалює весь газ).

1.4. Життєвий цикл смартконтракту

1. **Написання:** Створення .sol файлу.
2. **Компіляція:** Компілятор генерує:
   - **Bytecode:** Машинний код для EVM.
   - **ABI (Application Binary Interface):** JSON-файл, що описує методи контракту для взаємодії з ним.
3. **Деплой (Розгортання):** Відправка Bytecode у мережу через спеціальну транзакцію. Контракт отримує власну адресу.
4. **Взаємодія:** Виклик функцій через транзакції (зміна стану) або запити (читання).

Стандарти:

- **ERC-20:** Стандарт для взаємозамінних токенів (криптовалюти).
- **ERC-721:** Стандарт для NFT (унікальні активи).

2. Практична частина

Інструментарій

Ми будемо використовувати Remix IDE — потужне онлайн-середовище, яке не потребує встановлення.

Завдання: Смартконтракт "SimpleCounter"

Вам необхідно реалізувати контракт лічильника, який дозволяє зберігати число, змінювати його лише власнику та відстежувати зміни через події.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

/**
 * @title SimpleCounter
 * @dev Контракт для навчання основам Solidity
 */
contract SimpleCounter {
    // 1. Змінні стану (Storage)
    uint256 private count;
    address public owner;

    // Подія для логування змін
    event CountChanged(uint256 newValue, address changedBy);

    // Конструктор - виконується один раз при деплої
    constructor() {
        owner = msg.sender; // msg.sender - адреса того, хто розгортає контракт
        count = 0;
    }

    // 2. Власний модифікатор доступу
```

```
    modifier onlyOwner() {
        require(msg.sender == owner, "You are not the owner");
        _; // Продовження виконання функції
    }

    // 3. Функція для запису даних (Платна транзакція)
    function increment() public {
        count += 1;
        emit CountChanged(count, msg.sender);
    }

    // Функція з параметром та модифікатором
    function setCount(uint256 _newValue) public onlyOwner {
        count = _newValue;
        emit CountChanged(count, msg.sender);
    }

    // 4. Функція для читання (View - безкоштовно при виклику ззовні)
    function getCount() public view returns (uint256) {
        return count;
    }
}
```

Інструкція для виконання:

1. Відкрийте **Remix IDE**.
2. Створіть файл SimpleCounter.sol та вставте код вище.
3. Перейдіть у вкладку **Solidity Compiler** та натисніть "Compile SimpleCounter.sol".
4. Перейдіть у вкладку **Deploy & Run Transactions**:

- Environment: Виберіть **Remix VM (Cancun)** — це локальний симулятор блокчейну.
- Натисніть кнопку **Deploy**.

5. **Аналіз Gas Estimation:**
    - Після деплою розгорніть термінал знизу.
    - Викличте функцію increment. Подивіться на поле "Transaction Cost" (витрати газу на зміну стану).
    - Викличте getCount. Зверніть увагу, що це "Call" (не транзакція), і газ не списується з вашого балансу в симуляторі.

3. Контрольні запитання

1. Чому функції типу view не потребують оплати газом при виклику ззовні мережі (наприклад, через браузер), але споживають газ, якщо їх викликає інший смартконтракт під час транзакції?
2. Яка роль ABI у взаємодії між веб-сайтом (Frontend) та смартконтрактом? Чи можна взаємодіяти з контрактом, маючи лише його адресу, але без ABI?
3. Чим відрізняється ключове слово memory від storage у Solidity? Як вибір між ними впливає на вартість газу?
4. Що станеться з транзакцією, якщо в процесі виконання закінчиться газ (Out of Gas)? Чи повернуться кошти користувачу?
5. Поясніть механізм захисту від атак повторного входу (Reentrancy) на високому рівні. Навіщо використовувати require перед зміною балансів?