LAB 10 August, 2023

Events

1 Introduction

A fundamental feature that enables efficient and responsive **communication** within contracts.

Events play a crucial role in informing external applications about the state changes in contracts, facilitating the execution of dependent logic.

2 Purpose and Characteristics of Events:

Solidity events serve as **transaction logs** stored on the Ethereum Virtual Machine (EVM). They offer the following characteristics:

Responsiveness: Applications, such as Ether.js, can subscribe and listen to events through the RPC interface. By doing so, they can promptly respond to events at the frontend, enabling real-time updates and interactions with the contract.

Economical Data Storage: Storing data in events is cost-effective, consuming approximately 2,000 gas per event. In contrast, storing new variables on-chain requires at least 20,000 gas. Leveraging events can lead to significant gas savings and optimize contract performance.

3 Declaring and Defining Events:

Events are declared using the **event** keyword, followed by the event name and the type and name of each parameter to be recorded.

For example, let's consider the Transfer event from an ERC20 token contract:

```
event Transfer(address indexed from, address indexed to, uint256 value);
```

In this event declaration, ‘three’ parameters (from, to, and value) are recorded.

The **indexed** keyword is used to mark certain parameters for efficient querying.

4 Emitting Events:

Events can be emitted within functions to record specific occurrences.

Each time an event is emitted, the associated parameters are recorded in the transaction logs.

For instance, the following example demonstrates emitting a Transfer event within the _transfer() function:

```
 // define _transfer function,  execute transfer logic

    function _transfer(address from, address to, uint256 amount) external
{

        _balances[from] = 10000000; // give some initial tokens to
transfer address

        _balances[from] -=  amount; // "from" address minus the number of
transfer

        _balances[to] += amount; // "to" address adds the number of
transfer

        // emit event

        emit Transfer(from, to, amount);             }
```

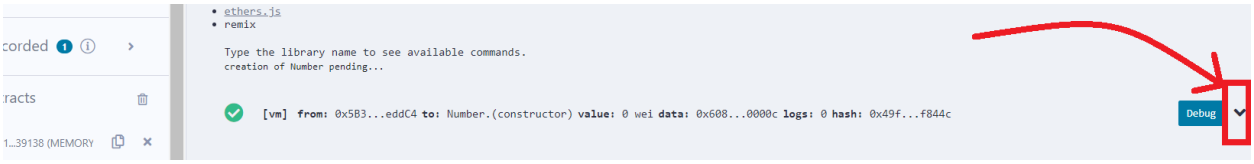
By emitting events, you can provide crucial information to external applications and enable them to react accordingly.

EXERCISE

Deploy the above smart contract and obtain the logs in json format.

Hint:

You can access the logs by following method:



EXERCISE

1) Deploy, observe and explain the following code.
2) Attach ScreenShot of Javascript log.
3) Create another event and emit your **'Name', 'ContractAddress'** and **'Department'.**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract Event {
    // Event declaration
    // Up to 3 parameters can be indexed.
    // Indexed parameters helps you filter the logs by the indexed
parameter
    event Log(address indexed sender, string msg);
    event AnotherLog();

    function test() public {
        emit Log(msg.sender, "Hello Web3.0");
        emit Log(msg.sender, "It's all about Decentralization");
        emit AnotherLog();
    }
}
```

5.5 EVM Log and Structure:

Under the hood, the EVM stores events using a data structure called a log.

Each log comprises two components

1) **Topics**: Topics describe events and include a maximum of four entries. Typically, the first topic is the event hash, calculated as the hash of the event signature. The remaining topics may contain indexed parameters for efficient querying.
2) **Data**: Non-indexed parameters are stored in the data section of the log. While they cannot be directly retrieved, the data section allows for storing larger and more complex data structures, such as arrays and strings, at a lower gas cost compared to topics.

It's worth noting that events facilitate efficient querying and enable applications to react to specific events of interest.

EXERCISE

1) Create a **struct** named 'PersonInfo', having multiple elements giving information about a person. Also, make a variable named **P1**, for that struct.

```
contract PersonContract {

    struct PersonInfo {...}

 PersonInfo public P1;
```

2) Make an **event** having parameters 'OldValue' of **PersonInfo** data type, NewValue of **PersonInfo** data type, 'timestamp' of **uint** data type and 'number' also of **uint** data type. Name the **event** as 'Update'.

```
 event Update(
        PersonInfo oldValue,
        PersonInfo newValue,
        uint256 timestamp,
        uint256 blockNumber
    );
```

3) Make a ‘setter’ function for your **P1** variable. The ‘setter’ function should be such that it also **emits** the ‘old value’ of **P1,** the ‘new value’ of **P1**, ‘timestamp’ of the block to show when it is emitted and ‘number’ of the block to show in which block number it is emitted.

```
function setPersonInfo(...) public {
        PersonInfo memory oldValue = P1;
        . . .

        emit Update(oldValue, P1, block.timestamp, block.number);
}
```

4) Deploy the contract and Observe the Logs.

EXERCISE

Create a **contract** that **emits** an **event** when the user assigns a **zero address** (an address consisting of only zeros) to the ‘owner’ state variable (**address** data type) on deployment of the contract.

Hint:

- Your **event** can have only one parameter of **string** type that **emits** a statement like “Zero Address inputted”.

```
event ZeroAddressInputted(string message);
```

- The statement states ‘on deployment of contract’, so you will make a ‘constructor’ that takes input from the user.

```
constructor(address _initialOwner)
```

- Using **if** condition, you can check whether user input is a **zero address** in the ‘constructor’ function.
- Inside the **if** condition you can **emit** your **event.**

```
emit ZeroAddressInputted("Zero Address inputted on deployment of contract");
```

EXERCISE

1) Make a **contract** named ‘NoPrime’. Make a **uint** type state variable ‘num’, and an **event,** having only two parameters of **string** and **uint** type.

```
contract NoPrime {

    uint public num;

    event PrimeChecker(string, uint);
```

2) Create a ‘**setter()**’ function for the ‘num’ variable that takes input from the user.

```
function setter(uint _inputNum) public
```

3) Make the **‘check()**’ function such that whenever the user inputs a ‘prime number’, it **emits** a statement informing that a prime number has been inputted along with the prime number itself.

```
function check(uint number) public pure {

      if (PrimeNo(number))

            emit PrimeChecker(The Number is Prime, number)

      else

            emit PrimeChecker(The Number is not Prime, number)
```

Hint:

- You can make a function that checks whether the number is prime or not and returns a **bool** value depending on the answer.
- You can then call that function inside the ‘check’ function and using the **if** condition you can check whether the ‘num’ is prime or not.