BlockBloom

Assignment – 1

Aarav Oswal – 230012

1.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract Factorial{

    function factorial(uint x) public pure returns(uint){

        uint result = 1;

        for(uint i=1; i<=x; i++)

        {

            result = result \* i;

        }

        return result;

    }

    function factRec(uint x) public pure returns(uint){

        if(x==0 || x==1)

        {

            return 1;

        }

        return x \* factRec(x-1);

    }

}

When I called both the functions for x=15, I got the gas fees of ‘factorial’ as 7047 gas and the gas fees of ‘factRec’ as 8056 gas. The recursive function is costlier to call as compared to the normal function.

2. Modifiers in Solidity:

In Solidity, modifiers are reusable code snippets that can be attached to functions to alter their functionality. They enable the implementation of restrictions, checks, and validations before executing a function's main task. This enhances code reusability, readability, and maintainability.

Types of Modifiers in Solidity:

1. Visibility Modifiers:

Visibility modifiers control who can access a specific function or state variable in a Solidity contract by defining its accessibility scope.

1. public: Accessible from anywhere, including external accounts, other contracts, and within the same contract.
2. private: Accessible only within the contract where it is defined.
3. internal: Accessible within the contract and its derived (inherited) contracts.
4. external: Accessible only from external accounts or other contracts, not within the contract itself.
5. Mutability Modifiers:

Mutability modifiers define whether a function can alter the contract's state or interact with Ether, specifying its behavior concerning the blockchain state.

1. view: The function can read state variables but cannot modify them.
2. pure: The function cannot read or modify state variables.
3. payable: The function can accept Ether transactions.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract Factorial{

    address owner;

    modifier onlyOwner{

        require(msg.sender == owner,"Only owner can perform his task");

        \_;

    }

    constructor(){

        owner = msg.sender;

    }

    function changeOwner(address newOwner) public onlyOwner{

        owner = newOwner;

    }

    function factorial(uint x) public onlyOwner view returns(uint){

        uint result = 1;

        for(uint i=1; i<=x; i++)

        {

            result = result \* i;

        }

        return result;

    }

    function factRec(uint x) public onlyOwner view returns(uint){

        if(x==0 || x==1)

        {

            return 1;

        }

        return x \* factRec(x-1);

    }

}

3. Solidity's Error Handling Mechanisms:

Error handling in Solidity is essential to ensure that smart contracts function correctly and can gracefully manage unexpected conditions. Solidity provides three main keywords for error handling: require, assert, and revert. Each serves a distinct purpose and is suited for different scenarios.

1. require:

Validates conditions such as user inputs, contract state, or external calls to ensure they meet specified criteria. Commonly used for input validation, access control, and external contract interactions.

If the condition evaluates to false, the transaction is reverted, undoing all state changes made during the execution. Optionally allows a custom error message to explain the failure.

Any remaining gas is refunded to the caller.

1. assert:

Used to catch internal errors and verify invariants (conditions that must always hold true). Typically used to detect programming bugs or logic errors that should never occur.

If the condition evaluates to false, the transaction is reverted, and all changes are rolled back. Does not support custom error messages.

All remaining gas is consumed, making it expensive when the condition fails.

1. revert:

Explicitly reverts a transaction, often with a custom error message. Useful for handling more complex error scenarios, such as nested conditions or dynamic error messages.

Stops execution and reverts all state changes. Can be invoked with or without a custom error message.

Remaining gas is refunded to the caller.

4. The self-destruct function in Solidity is used to destroy a smart contract and transfer its remaining Ether balance to a specified address. Once a contract is destroyed, it can no longer be interacted with.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.25;

contract Bank {

    mapping(address => uint) private balanceLedge;

    address public owner;

    modifier balanceCheck(uint amt) {

        require(balanceLedge[msg.sender] >= amt, "Insufficient Balance");

        \_;

    }

    modifier onlyOwner() {

        require(msg.sender == owner, "Only the owner can perform this action");

        \_;

    }

    constructor() {

        owner = msg.sender;

    }

    function deposit() public payable {

        balanceLedge[msg.sender] += msg.value;

    }

    function withdraw(uint \_amt) public balanceCheck(\_amt) {

        balanceLedge[msg.sender] -= \_amt;

        payable(msg.sender).transfer(\_amt);

    }

    function getBalance() public view returns (uint) {

        return balanceLedge[msg.sender];

    }

    function transfer(uint \_amt, address recipient) public balanceCheck(\_amt) {

        balanceLedge[msg.sender] -= \_amt;

        balanceLedge[recipient] += \_amt;

    }

    function closeBank(address payable recipient) public onlyOwner {

        selfdestruct(recipient);

    }

}

closeBank Function:

Calls the self-destruct opcode to destroy the contract.

Transfers any remaining Ether in the contract to the specified recipient address.

Effect on Smart Contract Behavior:

The code and storage of the contract are deleted from the blockchain.

Any interactions with the contract do not exist anymore.

Ether Transfer:

The contract's remaining Ether is sent to the address specified in selfdestruct.

Gas Refund:

There’s a gas refund for clearing the contract’s storage and code from the blockchain.

selfdestruct Status Update

EIP-4758 Proposal:

There is a current Ethereum community discussion on the possible deprecation of the selfdestruct opcode.

Reasons include potential misuse by intermediaries and challenges for state management in blockchain systems.

Alternatives:

Rather than calling selfdestruct developers are encouraged to provide mechanisms to disable or lock up contract functionality whilst not losing state.

5. The difference between the two functions transferEther() and display() is their visibility modifier. The visibility modifier of transferEther() is payable and that of display() is view. This means that the transferEther() function an receive Ether as part of its execution. It modifies the contract’s state, i.e it is a Transaction. The display() function on the other hand can only read data, it cannot modify the blockchain state. It is a call, not a transaction. A call returns output immediately because it does not invlove state changes. Transactions on the other hand, generate a transaction receipt which can be viewed in Remix console. It does not display output directly because transactions are asynchronous and are finalized only after confirmation on thr blockchain.

In general, output is displayed on the Remix IDE when the function is of the type view or pure as they are read-only. No output is displayed in case of functions without view or payable modifier especially payable functions.