BlockBloom Assignment 3

1. Write a solidity smart contract to find factorial of a number, write two functions with following function signature:
    - factorial(uint x) => returns factorial of x.
    - factRec(uint x) => returns factorial of x by recursion.
    - Compare and comment on the gas fees of both the functions.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.26;

contract FindFactorial {
    function factorialIterative(uint x) public pure returns (uint) {
        uint val = 1;
        for (uint i = 1; i <= x; i++) {
            val *= i;
        }
        return val;
    }
    function factRec(uint n) public view returns (uint) {
        if (n == 0 || n == 1) {
            return 1;
        } else {
            return n * factRec(n - 1);
        }
    }
}
```

The recursive method incurs greater Gas Fees because Calls itself repeatedly, each recursive call adding stack overhead.

2. What are modifiers in Solidity? Explore visibility modifiers and mutability modifiers in brief.
   Resubmit your code of Question 1 by adding an ‘onlyOwner’ modifier to factRec() function. Add another function that can be used to change the owner. (Recall that we had this modifier in the contracts we generated using OpenZeppelin last week)

- Modifiers in Solidity are special functions used to change or add preconditions, postconditions, or validations to other functions in a contract.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.26;

contract FindFactorial {
    address private owner; // Variable to store the owner address
    constructor() {
```

```
        owner = msg.sender;
    }
    modifier onlyOwner() {
        require(msg.sender == owner, "Only Owner can call.");
        _;
    }
    function changeOwner(address newOwner) public onlyOwner {
        require(newOwner != address(0), "Provide a Non zero address.");
        owner = newOwner;
    }
    function factorialIterative(uint x) public pure returns (uint) {
        uint val = 1;
        for (uint i = 1; i <= x; i++) {
            val *= i;
        }
        return val;
    }
    function factRec(uint n) public view onlyOwner returns (uint) {
        if (n == 0 || n == 1) {
            return 1;
        } else {
            return n * factRec(n - 1);
        }
    }
}
```

3. Explore error handling mechanisms in Solidity. In particular, look up what the keywords ‘require’, ‘assert’ and ‘revert’ are meant for.

- Require keyword is basically a check condition. Like
  require (msg.sender == owner, "Caller is not owner") checks whether the person who has called the function is owner or not, if not it throws up the error written there.
- Revert keyword is basically if statements, for eg if the message sender is not owner then we can throw error.
  if (msg.sender != owner){
  revert("Caller not Owner.");
  }
- Assert keyword is there to prevent / report the internal errors like invariant , overflow or underflow.
-
- Implement self-destruct in the bank contract discussed in last class. How is it affecting the smart contract behaviour ? Explore the self destruct opcode. Study the security implications and comment on its current status.
  - The contract ceases to exist on the blockchain. Any interaction with the contract's address (e.g., calling a function or querying state variables) will revert or fail. There are plans to phase out or restrict selfdestruct due to its potential misuse and unintended consequences.

```
    function deleteContract(address payable recipient) public onlyOwner {
        selfdestruct(recipient);
    }
```

4. Consider the following function:

```
function transferEther() public payable returns (address, uint)
{
    return (msg.sender, msg.value);
}
```

- When you execute the function, the output is not shown in the Remix IDE (just below the function call button).
- Contrast it with the behaviour of:

```
function display() public view returns (address)
    {
        return msg.sender;
    }
```

- Why is this happening? In general, when is the output displayed on the Remix IDE and when is it not?
- Hint: Observe the terminal carefully in each case.

ANSWER :

- The view modifier is used when the state of the blockchain is not changed and just information is read from it whereas payable is when transfer of ethers is allowed and therefore state of the blockchain can be changed and the change.
- In REMIX IDE, when we use the view function, since it is a call, the output is computed locally and returned directly to the caller.
- Remix IDE displays the return value immediately below the function call button because no transaction needs to be mined, and the return value is directly available.
- In the first case (transferEther), the terminal logs details about the transaction, such as:
    - Transaction hash: Unique identifier for the transaction.
    - Gas used: Amount of gas consumed for the transaction.
    - Value sent: Amount of Ether sent with the transaction.
- In the second case (display), the terminal directly shows the returned value because it's computed locally.