

# LAB 22 August, 2023

## 1 Interface

### 1.1 Introduction

An interface in Solidity is a way to define a set of function declarations without providing their implementation.

It is similar to an **abstract** contract, created using the '**interface**' keyword.

Interfaces allow contracts to interact with each other by providing a common set of functions that participating contracts must implement.

### 1.2 Rules and Principals

- 1) An interface cannot contain any function with an implementation. It only consists of function declarations, specifying the function name, input parameters, and return types. The purpose of an interface is to define a **common API** that implementing contracts must adhere to.
- 2) Functions defined in an interface can only be of type **external**. This means that the functions can be called from external contracts or transactions, but not from within the contract itself.
- 3) Interfaces cannot have constructors. Since interfaces do not contain any state variables, there is no need for constructors. Interfaces are purely focused on defining the external behavior of contracts.
- 4) Interfaces do not allow the declaration of state variables. They solely serve the purpose of defining **function signatures** and establishing a contract's external interface.
- 5) However, interfaces can have other elements like **enums and structs**. These additional elements can be accessed using the dot notation, where the interface name is followed by the element name.

### 4.3 Code Example

```
interface InterfaceExample{  
    // Functions having only declaration not definition  
    function getStr() external view returns(string memory);  
    function setValue(uint _num1, uint _num2) external;  
    function add() external view returns(uint);  
}
```

## 4.4 Practical Benefits

- Interfaces are extensively used in Solidity for contract **interaction and interoperability**.
- By defining interfaces, contracts can provide a standardized way for other contracts or **external entities** to interact with them.
- This promotes code **reusability**, as multiple contracts can implement the same interface and be used interchangeably.
- It's important to note that contracts implementing an interface must provide the exact **function signatures** specified in the interface. Failure to do so will result in a compilation error.
- Additionally, contracts can inherit multiple interfaces, allowing them to fulfill the requirements of multiple contracts simultaneously.

```
interface IERC20 {  
  
    event Transfer(address indexed from, address indexed to, uint256 value);  
  
    event Approval(address indexed owner, address indexed spender, uint256 value);  
  
  
    function totalSupply() external view returns (uint256);  
  
    function balanceOf(address account) external view returns (uint256);  
  
    function transfer(address to, uint256 amount) external returns (bool);  
  
    function allowance(address owner, address spender) external view returns (uint256);  
  
    function approve(address spender, uint256 amount) external returns (bool);  
  
    function transferFrom(address from, address to, uint256 amount) external returns (bool);  
}
```

## TASK

- 1) Observe and deploy the following smart contract.
- 2) Can we deploy the interface named '**Calculator**'?

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.17;

interface Calculator {

    function getResult() external view returns(uint);

}

contract Temp is Calculator {

    constructor() {}

    function getResult() external pure returns(uint result){

        uint a = 2;

        uint b = 5;

        result = b / a;

    }

}
```

## TASK

Observe and deploy the following smart contracts.

### Hint:

While using functions of '**MyContract**', input the contract address obtained on deploying '**Counter**' contract.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.17;

contract Counter {

    uint public count =2;

    function square() external {

        count = count ** 2;    }

}

interface ICounter {

    function count() external view returns (uint);

    function square() external;

}

contract MyContract {

    function squareCounter(address _counter) external {

        ICounter(_counter).square();    }

    function getCount(address _counter) external view returns (uint) {

        return ICounter(_counter).count();    }

}

```

## TASK

Define an interface **"Token"** with a function **"transfer"** that takes an address and an amount as parameters. Create a contract named **"TokenContract"**.

**"TokenContract"** should implement the "Token" interface and provide the logic for the "transfer" function.

```
interface Token {

    function transfer(address to, uint256 amount) external;    }

contract TokenContract is Token {

    . . .    }

```

## 2 Payable

### 2.1 Introduction

In the context of Solidity programming language, a **'payable'** function is a special type of function that enables a smart contract to accept **Ether**, the native cryptocurrency of the Ethereum blockchain.

Payable functions play a crucial role in facilitating the receipt of Ether and allowing developers to implement specific actions based on these incoming transactions.

By designating a function as payable using the keyword **"payable,"** we can indicate their expectation that Ether will be sent to that particular function within the smart contract.

#### Note:

“Failing to implement the necessary code to account for such deposits could result in Ether being locked forever or inaccessible to its intended recipient.”

### 2.2 EVM's Native Mechanism

To send Ether to a smart contract, the Ethereum Virtual Machine (EVM) provides a native mechanism. When someone initiates an Ether transfer to a smart contract, they do so by specifying a value in the transaction's value field.

In a JSON representation of a transaction, the structure would resemble the following:

```
{  
  "to": "0x5baf84167cad405ce7b2e8458af73975f9489291",  
  "value": "0xb1a2bc2ec50000", // 1 ether  
  "data": "0xd0e30db0" // deposit()  
  // ... other properties  
}
```

The behavior of the smart contract function upon receiving Ether depends on whether it is marked as payable or non-payable

## 2.3 Payable vs. Non-Payable Functions

### 1) Payable Function:

- If a function is declared as payable, it signifies that it can handle incoming Ether.
- When a transaction invokes a payable function, the associated logic within the function will be executed.
- This allows developers to perform operations such as logging an event, modifying storage to record the deposit, or even reverting the transaction if necessary.

```
function deposit() public payable {}
```

Notice that, writing a payable function alone is enough to receive ether and you may not need to write any logic.

### 2) Non-Payable Function:

- If a function is not marked as payable, any transaction attempting to send Ether to that function will be reverted
- The funds will be returned to the sender, excluding the gas cost incurred for the transaction.

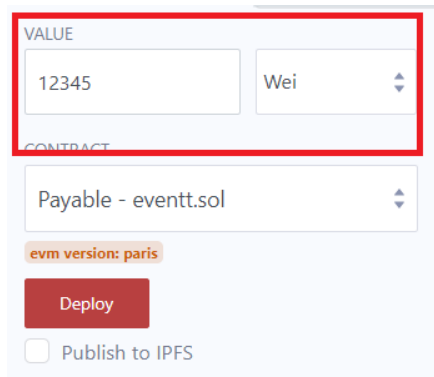
```
function notPayable() public {}
```

## Conclusion

- Payable functions in Solidity serve as a critical mechanism for smart contracts to accept Ether and manage incoming transactions effectively.
- By explicitly labeling functions as **payable**, we ensure that the smart contract accounts for and processes incoming Ether appropriately, preventing potential issues such as funds being locked or inaccessible.

## TASK

- 1) Deploy, observe and explain the following code.
- 2) Call each function along with some Ether, and then explain difference between both functions



- 3) Observe the difference in gas fees for both functions

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

contract Payable {
    address payable public owner;

    constructor() payable {
        owner = payable(msg.sender);
    }
    function deposit() public payable returns (uint account, uint cont){
        cont = address(this).balance;
        account = owner.balance;
    }
    function notPayable() public view returns (uint account, uint cont){
        cont = address(this).balance;
        account = owner.balance;
    }
}
```

## TASK

Create a simple Solidity contract named **'PaymentContract'** with a payable function called **'receivePayment'**.

The function should accept incoming Ether and emit an event indicating the sender's address and the amount of Ether received.