5️⃣

Basics V: Functions

Function Modifiers

Function Modifiers are used to modify the behaviour of a function.

At first, create a modifier with or without a parameter as shown below.

```
contract Owner {
   address owner;
   constructor() public {
      owner = msg.sender;
   }
   modifier onlyOwner {
      require(msg.sender == owner);
      _;
   }
   modifier costs(uint price) {
      if (msg.value >= price) {
         _;
      }
   }
}
```

The function body is inserted where the special symbol "_;" appears in the definition of a modifier. So if the condition of the modifier is satisfied while calling this function, the function; otherwise and otherwise, an exception is thrown.

Complete your program. See the example below.

```
contract Register is Owner {
   mapping (address => bool) registeredAddresses;
   uint price;
   constructor(uint initialPrice) public { price = initialPrice; }

   function register() public payable costs(price) {
      registeredAddresses[msg.sender] = true;
   }
   function changePrice(uint _price) public onlyOwner {
      price = _price;
   }
}
```

Events

An event is an inheritable member of the contract, which stores the arguments passed in the transaction logs when emitted.

Generally, events are used to inform the calling application about the current state of the contract, with the help of the logging facility of EVM.

Events notify the applications about the change made to the contracts and applications which can be used to execute the dependent logic.

`event <eventName>(parameters) ;`

> **We can add atmost 3 indexes in one event.**

An event can be called from any method by using its name and passing the required parameters.

Sending & Receiving Ether in Solidity

How to Send Ether?

1. **transfer:**
   - has a gas limit of 2300
   - throws error
   - needs a fallback function in the contract to be sent ether
   - no longer recommended

```
function depositUsingTransfer(address payable _to) public payable {
  _to.transfer(msg.value); //2300 gas
}
```

2. **send:**
   - similar to `transfer` method
   - has a gas limit of 2300
   - returns the status as a boolean
   - no longer recommended

```
function depositUsingSend(address payable _to) public payable {
  bool isSent = _to.send(msg.value); //2300 gas
  require(isSent, "The transaction is failed.");
}
```

3. **call:**
   - the **recommended** way of sending ETH to a smart contract
   - forward all gas or set a fixed amount
   - the empty argument triggers the ***fallback*** function of the receiving address
   - can also trigger other ***functions*** defined in the contract and send a fixed amount of gas to execute the function
   - the transaction status is sent as a **boolean** and the return value is sent in the data variable
   - `call` in combination with re-entrancy guard is the recommended method to use after December 2019. Guard against re-entrancy by
     - making all state changes before calling other contracts
     - using re-entrancy guard modifier

```
function depositUsingCall(address payable _to) public payable {
  (bool isSent, ) = _to.call{value: msg.value}("");
  require(isSent, "The transaction is failed.");
}
```

How to Receive Ether?

```
flowchart TD
    A[Send Ether] --> B{Is msg.data empty?}
    B -- Yes --> C{"Does receive() exist?"}
    C -- Yes --> D["receive()"]
    C -- No --> E["fallback()"]
    B -- No ----> E["fallback()"]
```

A contract receiving Ether must have at least one of the functions below

- `receive() external payable`

- `fallback() external payable`

`receive()` is called if `msg.data` is empty, otherwise `fallback()` is called.

The Fallback Function

- Does not have a name
- Does not take any inputs
- Does not return an output
- Must be declared as `external`

> Do not write too much code into fallback function. Reduce gas cost as much as possible, otherwise the function fails.

```
contract Receive {

    event Log(uint _gas);

    receive() external payable {}

    fallback() external payable {
        emit Log(gasleft());
    }

    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}

contract Send {

    function useTransfer(address payable _to) public payable {
        _to.transfer(msg.value);
    }

    function useSend(address payable _to) public payable {
        bool isSent = _to.send(msg.value);
        require(isSent, "Transaction failed! Ether couldn't be sent.");
    }

    function useCall(address payable _to) public payable {
        (bool isSent, ) = _to.call{value: msg.value}("");
        require(isSent, "Transaction failed! Ether couldn't be sent.");
    }
}
```

Function Overloading

Having multiple definitions for the same function name in the same scope is allowed in Solidity.
The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

```
contract FunctionOverload {

    function sum(uint a, uint b) public pure returns (uint) {
        return a + b;
    }

    function sum(uint a, uint b, uint c) public pure returns (uint) {
        return a + b + c;
    }

    function getSumofTwo() public pure returns (uint) {
        return sum(1, 3);
    }
```

```
    function getSumofThree() public pure returns (uint) {
        return sum(1, 3, 5);
    }
}
```

Cryptographic Functions

A Cryptographic Hash Function (CHF) is a mathematical algorithm that maps data of arbitrary size, often called *message* , to an array of fixed size (*hash*, *hash value*, *message digest*).

It's a one-way function, that is, a function which is practically infeasible to invert or revert the computation.

Solidity provides important built-in cryptographic functions:

- **keccak256 (bytes memory) returns (bytes32):** Computes keccak256 hash of the input
- **sha256 (bytes memory) returns (bytes32):** Computes sha256 hash of the input
- **ripemd160 (bytes memory) returns (bytes20):** Computes ripemd160 hash of the input

Keccak is a leading hash function designed by non-NSA designers. *Keccak* is a family of cryptographic sponge functions.

```
contract Cryptographic {

    function generateHash() external view returns (uint) {
        return uint((keccak256(abi.encodePacked(block.timestamp, block.number, block.difficulty))));
    }

    function generateRandom(uint _range) external view returns (uint) {
        return ((uint(keccak256(abi.encodePacked(block.timestamp, block.number, block.difficulty))) % _range));
    }
}
```

In the example above, we needed dynamically changing values such as timestamp, block number and difficulty to generate random numbers. `abi.encodePacked` concatenates the arguments nicely.

The code above is open to manipulation. **WHY?**

Oracles

Oracles serve as a bridge between the blockchain and the real world. Real-world information is provided for smart contracts via oracles.

```
contract Oracle {
    address admin;
    uint public rand;

    constructor () {
        admin = msg.sender;
    }

    function feedRand(uint _rand) public {
        require(msg.sender == admin, "The caller is not admin.");
        rand = _rand;
    }
}

contract Cryptographic {

    Oracle oracle;

    constructor (address oracleAddress) {
        oracle = Oracle(oracleAddress);
    }

    function generateHash() external view returns (uint) {
        return uint((keccak256(abi.encodePacked(block.timestamp, block.number, block.difficulty))));
    }
```

```
    function generateRandom(uint _range) external view returns (uint) {
        return ((uint(keccak256(abi.encodePacked(oracle.rand(), block.timestamp, block.number, block.difficulty))) % _range));
    }
}
```

The code above is not a great example of oracles, but it shows the essential logic of oracles.