Lesson 17: PROGRAMMING SMART CONTRACTS: SOLIDITY

28/04/2021
PROGRAMMING SMART CONTRACTS: SOLIDITY

- a contract oriented high level language for the EVM
  - static type system
  - syntax very similar to javascript

- a language in continuos evolution
  - last version v0.8.4
  - https://docs.soliditylang.org/en/v0.8.4/

- in the next slides an example-based tutorial
  - a Ponzi scheme
  - a lottery
  - Satoshi dice

- use on-line docs and Remix, a web based interface, to have more insights in the language
A PONZI SCHEME

• in brief
  • take investments from investors
  • give back the return to the early investors
    with the money received from new investors
  • needs a constant flux of new investors
  • the crash
    • eventually, there are not enough funds from
      new investors to support the scheme, causing it to crash

• some famous Ponzi Schemes
  • the first one: Charles Ponzi
  • another famous one: Bernie Madoff, in 2008
A SIMPLIFIED PONZI SCHEME

• a simplified Ponzi Scheme is the first, naive Solidity program

  • for each new investment, take the money and send it to the previous investor

  • each investment must be larger than the previous one: check if this true, otherwise discard the investment

  • each investor, except the last one, will get a return on their investment
pragma solidity ^0.5.2;

contract SimplePonzi {

    address payable currentInvestor; uint public currentInvestment;
    bool notfirst;

    function investment () public payable {
        // new investments must be 10% greater than current
        uint minimumInvestment = currentInvestment * 11 / 10;
        require(msg.value > minimumInvestment);

        address payable previousInvestor = currentInvestor;
        currentInvestor = msg.sender;
        currentInvestment = msg.value;

        // payout previous investor
        if (notfirst)
        {
            previousInvestor.transfer(msg.value);
            notfirst=true;
        }
    }

    fallback ( ) payable external { investment(); }
}
THE PRAGMA DIRECTIVE

• the first line of code is a “version pragma”

    pragma solidity ^0.5.2;

• defines the range of versions of the compiler which can be used

    matches version of the compiler ≥ 0.5.2 and < 0.6.0

• the caret (^) before the version number points out that
  • Solidity can use the version of the compiler starting from 0.5.2 within the
    major build (5) range
    • future compiler versions might introduce incompatible changes
  • no caret specified: all the versions newer than the one referred are ok
  • more complex expressions can be specified to define versions ranges

• very useful for languages in high evolution, like Solidity
**ANALYSING SIMPLE PONZI**

```solidity
contract SimplePonzi {
    uint public currentInvestment = 0; bool notfirst;
    address payable currentInvestor;
}
```

- **contract** is like a class
- **currentInvestment**
  - amount of the investment that will be lost by the last investor if there will be no further investments
- **currentInvestor** is the address of the most recent investor
  - the only one that has not yet received a return on the investment
  - the sucker which will lose his/her investment if no other one will make an investment
contract SimplePonzi {
    address payable currentInvestor;
    uint public currentInvestment; bool notfirst;

    • address represents the public address of an EOA or of a smart contract
      • 20 byte value represented in hexadecimal prefixed with 0x.
      • some ways to assign values to addresses:

        //assign a fixed value
        address myAddress = 0xE0f5206BBD039e7b0592d8918820024e2a7437b9;

        //assign address of the issuer of the transaction/message
        address sender = msg.sender

        //sets the address to 0x0
        address emptyAddress = address(0)

        //assign the contract current address
        address current = address(this)
contract SimplePonzi {
    address payable currentInvestor;
    uint public currentInvestment; bool notfirst;

    • the address has several properties which can be queried:
      • address.balance the balance in Wei of the Address
      • address.code is the code at that address (may be empty)
    • address payable is an address you can send Ether to
pragma solidity ^0.6.8;

contract MyContract {
    address private owner;

    constructor() public{
        owner=msg.sender;
    }

    function getOwner() public view returns (address) {
        return owner;
    }

    function getBalance() public view returns(uint256){
        return owner.balance;
    }
}

msg.sender a variable in the global namespace
the address that sent the transaction containing the call to the function
contract SimplePonzi {
    address payable currentInvestor;
    uint public currentInvestment; bool notfirst;

    • integer types
      • int (signed)
      • uint (unsigned)
    • keywords: uint8 / int8 t- uint256 / int256 in step of 8.
      • various size to minimize gas consumption and storage space
    • uint / int alias for uint256 / int256.
    • operators as usual: comparison, arithmetic, bitwise, shift
    • division always results in an integer and rounds towards zero (5 / 2 = 2).
    • fixed point number not yet fully supported
A default value is defined for each type

<table>
<thead>
<tr>
<th>type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8-uint256</td>
<td>0 (zero)</td>
</tr>
<tr>
<td>int8-int256</td>
<td>0 (zero)</td>
</tr>
<tr>
<td>bool</td>
<td>false</td>
</tr>
<tr>
<td>string</td>
<td>An empty string</td>
</tr>
<tr>
<td>byte</td>
<td>In integer form, 0; in hex form, 0x00</td>
</tr>
<tr>
<td>bytes1-bytes32</td>
<td>In integer form, 0; in hex form, 0x&lt;NumberOfZeros&gt; (NumberOfZeros = NumberOfBytes * 2; for example, for bytes2, the default value is 0x0000)</td>
</tr>
<tr>
<td>bytes</td>
<td>Empty bytes prefixed with 0x</td>
</tr>
<tr>
<td>address</td>
<td>0x000000000000000000000000000000000000000000000</td>
</tr>
</tbody>
</table>

**SOLIDITY DATA TYPES**
FALLBACK FUNCTIONS

```solidity
fallback () payable external {
    investment();
}
```

- an unnamed function: at most one for each contract
  - no arguments, no returned values
- acts as a default function to be executed when
  - no other functions match the function referred in the call
  - if marked `payable`, when a transaction payment is sent to the contract, without an explicit function call by the sender
- useful for contracts with a single type of payment
  - in the Ponzi contract, a single transfer of money, for the investment
  - when a user sends money to the contract, the fallback function is invoked
- Solidity > 0.6 defines also
  - `receive( ) public payable { }` to receive ethers
PAYABLE FUNCTIONS

function investment () public payable {
    // new investments must be 10% greater than current
    uint minimumInvestment = currentInvestment * 11 / 10;
    require(msg.value > minimumInvestment);
}

• function is like a method

• visibility modifier
  • external
  • public
  • internal
  • private
FUNCTIONS VISIBILITY

- **public** can be called from other contracts, internally and from externally owned accounts.
  - no restrictions
- **internal**
  - only the current contract and contracts inheriting from it can execute the function
- **external**
  - can be triggered only by a transaction or by external contract message
  - if f is external, f() does not work, this.f() works.
- **private**
  - are accessible only from the contract where they are defined and not by derived contracts
FUNCTIONS VISIBILITY

- the terms internal and private are somewhat misleading

- any function or data inside a contract is always visible on the public blockchain
  - anyone can see the code or data.
  - the keywords described here only affect how and when a function can be called.

- why both public and external?
  - from the docs: external functions are sometimes more efficient when they receive large arrays of data.
PAYABLE FUNCTIONS

```solidity
function investment () public payable {
    // new investments must be 10% greater than current
    uint minimumInvestment = currentInvestment * 11 / 10;
    require(msg.value > minimumInvestment);
}
```

- transaction is like an envelope
  - contents of the letter are function parameters
  - adding a value to the transaction is like putting cash inside the envelope
- a payable function
  - to execute, it requires a payment in the transaction
  - the amount sent is taken from the `msg.value` field in units of wei
  - cryptocurrency sent is stored in the contract's account
  - if wei are sent to a not payable function, the transaction is rejected
function investment () public payable {
    // new investments must be 10% greater than current
    uint minimumInvestment = currentInvestment * 11 / 10;
    require(msg.value > minimumInvestment);

    • any new investment must be at least 10% greater than the current investment
    • to guarantee a juicy return to the investors
    • otherwise it will be rejected

    • no decimals in Solidity, so need to multiply by 11 and then divide by 10
    • require tests conditions on function arguments or transaction fields
      • throws an error and stops execution if some condition is not true
      • reverts all the changes made
      • consumes the gas up to the point of failure
      • refunds the remaining gas
ACCESSING TRANSACTION PROPERTIES

function investment () public payable {
    // new investments must be 10% greater than current
    uint minimumInvestment = currentInvestment * 11 / 10;
    require(msg.value > minimumInvestment);
}

• several special predefined variables and functions in the global namespace provide information on the transactions that has invoked a function
  • msg provides information contained in the transaction/message call
    • msg.sender sender of the call/of the message (address)
    • msg.value value sent in wei (uint) (the investment in our case)
    • msg.gas gas unused after the execution of the transaction
    • msg.data complete calldata (bytes)
Solidity provides access to global variables that are

- not declared within contracts
- accessible from code within contracts.

contracts cannot access the ledger directly.

- a ledger is maintained by miners only

however Solidity provides some information about the current transaction and block to contracts so that they can utilize them.

- block-related variables
- transaction-related variables.
TRANSACTION AND BLOCK GLOBAL VARIABLES

- `tx.gasprice` the gas price caller is ready to pay for each gas unit
- `tx.origin` the first caller of the transaction
- `block.timestamp`
- `block.number`
- `block.gaslimit`
- `block.difficulty`
- `block.coinbase`
address previousInvestor = currentInvestor;
currentInvestor = msg.sender;
currentInvestment = msg.value;

// payout previous investor
previousInvestor.send(msg.value);

- onboarding the new sucker
- keep a reference to the previous investor so to pay out him/her with the next investment
- three ways to send ether in Solidity:
  - address.transfer(value)
  - address.send(value)
  - address.call.value(value)()
TRANSFERRING ETHERS

- `address.transfer` (value)
  - transfer ether units in wei
  - throws an error if transfer fails, or other exceptions occurs, like out of gas
  - the most secure one

- `address.send` (value)
  - returns false if some failure happens
  - less secure: give responsibility to the users to manage the failure

- both of them trigger the receiving contracts' `fallback function`
  - the called function is only given a limited amount of 2,300 gas
  - to avoid improper use of these functions which have been the first source of Solidity bugs
    - the famous DAO attack to Ethereum
A MORE REALISTIC PONZI

• new investments are distributed evenly between all previous investors
  • after the distribution is complete, the newest investor is added to the list of investors
  • avoid adding the complexity of tracking investor shares
  • but...no incentive to send more than the minimum payment

• as the number of investors in the Ponzi scheme increases, the return for an investor from each new investment decreases.

• added a minimum investment
  • prevent freeloaders from sending a 0-value transaction to become an investor

• the creator gets the privilege of joining the Ponzi without having to send any ether.
pragma solidity ^0.6.0;

contract GradualPonzi {
    address [] public investors;
    mapping (address => uint) public balances;
    uint public constant MINIMUM_INVESTMENT = 1e15;
    constructor () public 
    {investors.push(msg.sender); }

    function investment () public payable {
        require(msg.value >= MINIMUM_INVESTMENT);
        uint eachInvestorGets = msg.value / investors.length;
        for (uint i=0; i < investors.length; i++)
            balances[investors[i]] += eachInvestorGets;
        investors.push(msg.sender); }

    function withdraw () public {
        uint payout = balances[msg.sender]; balances[msg.sender] = 0;
        msg.sender.transfer(payout);
    }

    fallback() payable external {investment();}
pragram solidity ^0.6.0;
contract GradualPonzi {
    address [] public investors;
    mapping (address => uint) public balances;
    uint public constant MINIMUM_INVESTMENT = 1e15;

    • fixed sized arrays
      • do not grow or shrink capacity
      • byte [ ]: some predefined fixed sized array: bytes1,...bytes32
      • length property
    • address[ ] investors dynamically sized arrays
      • push to append a new element to the last position of the array
      • length property
MAPPINGS

pragma solidity ^0.6.0;
contract GradualPonzi {
  address [] public investors;
  mapping (address => uint) public balances;
  uint public constant MINIMUM_INVESTMENT = 1e15;

  • mapping (Key_Type => Value_Type)
  • like hash tables
    • provide lookups and writes
    • Keccak256 of Key_Type
  • when the map is declared (before having actually written anything to it)
    • all possible addresses exist
    • every key implicitly bound to all-zero value
    • binding is always defined
pragma solidity ^0.6.0;

contract GradualPonzi {
    address [] public investors;
    mapping (address => uint) public balances;
    uint public constant MINIMUM_INVESTMENT = 1e15;

    • minimal investment
    • 1e15 = 1 finney

<table>
<thead>
<tr>
<th>Suffix example</th>
<th>In wei</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 wei</td>
<td>1</td>
</tr>
<tr>
<td>1 szabo</td>
<td>1e12</td>
</tr>
<tr>
<td>1 finney</td>
<td>1e15</td>
</tr>
<tr>
<td>1 ether</td>
<td>1e18</td>
</tr>
</tbody>
</table>
A LOTTERY ON THE BLOCKCHAIN

- run a lottery on the blockchain, no anchoring it in any single legal jurisdiction
- main problem
  - how to obtain a source of entropy in a deterministic environment?
- solutions may be to use:
  - an external oracle
    - add complexity and external dependencies
  - data taken from the blockchain
    - the blockhash as a source of randomness
    - be aware that miners can perform an attack, cheat and influence a value, for instance may change the timestamp of the block
- more complex solutions
  - RanDAO
  - Verifiable Random Functions, we will see these in Algorand
pragma solidity ^0.8.0;

contract SimpleLottery {
    uint public constant TICKET_PRICE = 1e15; // 1 finney
    address [] public tickets;
    address payable public winner;
    uint public ticketingCloses;

    constructor (uint duration) public {
        ticketingCloses = block.timestamp + duration;
    }

    function buy () public payable {
        require(msg.value == TICKET_PRICE);
        require(block.timestamp < ticketingCloses);
        tickets.push(msg.sender);
    }
}
A LOTTERY ON THE BLOCKCHAIN

function drawWinner () public {
    require(block.timestamp > ticketingCloses + 5 minutes);
    require(winner == address(0));
    bytes32 bhash=blockhash(block.number-1);
    bytes memory byteArray = new bytes(32);
    for (uint i; i <32; i++){
        byteArray[i] =bhash[i];
    }
    bytes32 rand=keccak256(byteArray);
    winner = payable(tickets[uint(rand) % tickets.length]);
}

function withdraw () public {
    require(msg.sender == winner);
    winner.transfer(address(this).balance);
}

fallback () payable external {
    buy(); }
}
constructor (uint duration) public

{ ticketingCloses = block.timestamp + duration; }

• constructor: put here the inizializations
  • how long does the lottery goes on?
  • during this period, people can buy tickets, afterwards stop selling tickets
• block.timestamp
  • current block timestamp
  • the timestamp of the block where the transaction is inserted
  • returns a UNIX timestamp: seconds after the UNIX epoch of 1970-01-01 00:00:00
  • makes it easy to create delayed actions
function buy () public payable {
    require(msg.value == TICKET_PRICE);
    require(block.timestamp < ticketingCloses);
    tickets.push(msg.sender);
}

fallback () payable public { buy(); }

- a user sends finneys to the contract, without calling any function
- fallback is activated, then it invokes the function buy()
- buy checks that the price is ok and that the lottery has not been stopped, if both conditions are true
  - the address of the sender is stored in the tickets array
  - the balance of the contract is automatically increased by price of the sold ticket (implicit, you do not see this in the code)
Solidity is a deterministic language

- all the validators (miners) must return the same result
- cannot use a built-in source of entropy to generate random numbers.

- a limited source of entropy

- take the hash of the previous block by the expression
  \(\text{blockhash}(\text{block.number - 1})\)
- apply the \texttt{keccak256} function

```solidity
function drawWinner () public {
    .......
    bytes32 bhash=blockhash(block.number-1);
    bytes memory byteArray = new bytes(32);
    for (uint i; i <32; i++){  byteArray[i] =bhash[i];}
    bytes32 rand   =keccak256(bytesArray);
    winner        = payable(tickets[uint(rand) % tickets.length]);
}
```
use the globally available variables

- `block.number`: current block number
- `blockhash(uint blockNumber)`
  - hash of a given block number
  - only works for the 256 most recent blocks

```solidity
function drawWinner () public {
    require(block.timestamp > ticketingCloses + 5 minutes);
    require(winner == address(0));
    bytes32 bhash=blockhash(block.number-1);
    bytes memory byteArray = new bytes(32);
    for (uint i; i <32; i++){  byteArray[i] =bhash[i];}
    bytes32 rand = keccak256(byteArray);
    winner = payable(tickets[uint(rand) % tickets.length]);
}
```
LOTTERIES: DRAWING THE WINNER

```solidity
function drawWinner () public {
    require(block.timestamp > ticketingCloses + 5 minutes);
    require(winner == address(0));
    ...
}
```

- set a time delay between the stopping of the ticket purchase period and the drawing of the winner
- why a delay?
  - the blockhash must be unguessable for the user when he/she buy a ticket
  - guarantees that no one can know the blockhash, while buying a ticket
  - avoid attacks
- ensure also that the winner has not already been drawn, when invoked
  - use address(0), the same as "0x0", an uninitialized address.
GAMBLING: SATOSHI DICE

- an early Bitcoin gambling service
  - "blockchain-based betting game" operating since 2012
- how to play?
  - send a transaction to one of the addresses operated by the service,
    - each address has a different payout
  - the service determines if the wager wins or loses, then sends a transaction in response with
    - the payout to a winning bet
    - a tiny fraction of the house's gain to a losing bet.

- responsible for half the transactions on the Bitcoin network, in the first period
- many alternate implementations exist today
GAMBLING: SATOSHI DICE

- implementation
  - submit a transaction with
    - a number from 0-65,535 ($2^{16} = 65,536$)
    - an amount of ether
  - the game generates a random number in the same range using a secret seed.
  - if the generated number is below the submitted number, the user wins money.
  - the amount of money won is dependent on the submitted number.
  - the lower the number, the higher the multiplier and payout
    - $32,000 = \sim 2x$, $16,000 = \sim 4x$
  - in the real implementations, periodically publish the hash of the old seeds together the betting addresses to provide auditability
pragma solidity ^0.8.0;

contract SatoshiDice {
    struct Bet {
        address user;
        uint block;
        uint num;
        uint amount;
    }
    uint public constant FEE_NUMERATOR = 1;
    uint public constant FEE_DENOMINATOR = 100;
    uint public constant MAXIMUM_NUM = 100000;
    uint public constant MAXIMUM_BET_SIZE = 1e18;
    address payable owner; address payable temp;
    uint public counter;
    mapping(uint => Bet) public bets;
    event BetPlaced(uint id, address user, uint cap, uint amount);
    event Roll(uint id, uint rolled);
constructor () public {
    owner = payable(msg.sender);
}

function wager (uint num) public payable {
    require(num <= MAXIMUM_NUM);
    require(msg.value <= MAXIMUM_BET_SIZE);
    counter++;
    bets[counter] = Bet(msg.sender, block.number + 3, num, msg.value);
    emit BetPlaced(counter, msg.sender, cap, msg.value);
}
```solidity
function roll(uint id) public {
    Bet storage bet = bets[id];
    require(msg.sender == bet.user);
    require(block.number >= bet.block);
    require(block.number <= bet.block + 255);
    // X=block.blockhash(bet.block)||id
    // this is a high level concatenation, not actual code
    bytes32 random=keccak256(X);
    uint rolled = uint(random) % MAXIMUM_NUM;
    if (rolled < bet.num) {
        uint payout = bet.amount * MAXIMUM_NUM / bet.num;
        uint fee = payout * FEE_NUMERATOR / FEE_DENOMINATOR;
        payout -= fee; temp=payable(msg.sender)
        temp.transfer(payout);
    }
    emit Roll(id, rolled); delete bets[id];
}
```
fallback () payable external {}

function kill () public {
    require(msg.sender == owner);
    selfdestruct(owner);
}

contract SatoshiDice {
    struct Bet {
        address user;
        uint block;
        uint cap;
        uint amount;
    }

    • similar to C struct
      • defines a complex data type that has other data types as members.
    • any data type can appear in a struct and nesting structs is permitted.
    • declaring a struct creates a constructor that can be used to instantiate instances of that struct.
    • struct members are accessed with the . notation (bet.line, bet.amount, etc.).
LOGGING EVENTS ON THE BLOCKCHAIN

```solidity
event BetPlaced(uint id, address user, uint cap, uint amount);

event Roll(uint id, uint rolled);
```

- **events**
  - BetPlaced wager placed
  - Roll wager resolved
- declared with the keyword `event` followed by the name of the event
  - syntax similar to `struct`

- **event logs:**
  - are registered on the blockchain
  - any JavaScript app in the frontend can listen for events as callbacks
  - events can be indexed: search for specific events in the log
function wager (uint cap) public payable {
    require(cap <= MAXIMUM_NUM);
    require(msg.value <= MAXIMUM_BET_SIZE);
    counter++;
    bets[counter] = Bet(msg.sender, block.number + 3, cap, msg.value);
    BetPlaced(counter, msg.sender, cap, msg.value);
}

• generate a new ID (counter) for this bet
  • registered as an event on the blockchain
  • the gambler can read the ID paired with his/her bet querying the blockchains
    • can later use the ID in the request to roll function

• locks the block number for the generation of random numbers
  • set a time forward of three blocks in the future, user
    • must wait 3 blocks after wagering
    • cannot guess the hashblock of 3 blocks in the future
ROLLING THE DICE

Function roll

```solidity
function roll(uint id) public {
    Bet storage bet = bets[id];
    require(msg.sender == bet.user);
    require(block.number >= bet.block + 3);
    require(block.number <= bet.block + 255);
    // X = block.blockhash(bet.block) || id
    // this is a high level concatenation, not actual code
    bytes32 random = keccak256(X)
}
```

- Function roll simulate the “dice roll”
- The user must trigger the roll within 255 blocks of the bet block
  - Otherwise, stop and throw an error
- Solidity stores only the 256 most recent blockhashes
  - Waiting longer will lead to blockhash of 0x0
DATA LOCATION IN SOLIDITY

```solidity
function roll(uint id) public {
    Bet storage bet = bets[id];
}
```

- **data location**
  - memory and storage
- **storage**
  - variables stored permanently on the blockchain. “like the hard disk”
  - stored in the state tree (Patricia Merkle trie) of the block
  - expensive and should be used only when necessary
- **memory**
  - local available to every function within a contract
  - temporary variables, “like the RAM”.
  - cleared after the function execution
  - cheap and should be used whenever possible.
fallback () payable public {} 

function kill () public {
    require(msg.sender == owner);
    selfdestruct(owner);  }

- fallback()
  - before starting fee gathering, the contract has to be funded
  - an initial amount collected by function fund() to allow starting bets payout

- function selfdestruct(address recipient)
  - destroys the contract
  - remaining balance, sent to the address passed as argument
Solidity supports contract inheritance between smart contracts

- **multiple inheritance is supported**

- **multiple contracts related by a parent-child relationship**

- **internal variables and functions are available to derived contract, according to the visibility rules**
pragma solidity ^0.6.0;

contract Flower {
    address owner;
    string flowerType;

    constructor (string memory newFlowerType) public {
        owner = msg.sender;
        flowerType = newFlowerType;
    }

    function water() public pure returns (string memory) {
        return "ohhhh, thanks, I love";
    }
}

contract Rose is Flower ("Rose") {
    function pick() public pure returns (string memory) {
        return "ouuuch";
    }
}

contract Jasmine is Flower ("Jasmine") {
    function smell() public pure returns (string memory) {
        return "Mmmmm, smells good!!";
    }
}
INHERITANCE IN REMIX

Environment: JavaScript VM
Account: 0xca3...a733c (99.9999999999981781)
Gas limit: 3000000
Value: 0 (wei)

Flower
- Jasmine
- Rose

Load contract from Address
At Address

0 pending transactions

Flower at 0x692...77b3a (memory)
INHERITANCE IN REMIX

- Flower at 0x692...77b3a (memory)
  - water

- Jasmine at 0xbbf...732db (memory)
  - smell
  - water

- Rose at 0x0dc...97caf (memory)
  - pick
  - water
PURE AND VIEW FUNCTIONS

- pure functions
  - do not read or modify the state
  - only use the parameters of the function
- view functions
  - do not modify the state

```solidity
pragma solidity ^0.6.0;

contract ViewAndPure {
    uint public x = 1;
    // Promise not to modify the state.
    function addToX(uint y) public view returns (uint) {
        return x + y;
    }
    // Promise not to modify or read from the state.
    function add(uint i, uint j) public pure returns (uint) {
        return i + j;
    }
}
```
DEFINING AN INTERFACE

- for your contract to interact to another contract on the blockchain
  - import the contract code OR
  - use an abstract contract/interface
- contract interface: like a contract skeleton, similar to JAVA interfaces
  - only functions declaration, no body
  - do not define the function bodies

```
pragma solidity ^0.6.0;

interface Calculator {
    function getResult() external view returns (uint);
}

contract Test is Calculator {
    constructor() public {} 
    function getResult() override external view returns (uint){
        uint a = 1;
        uint b = 2;
        uint result = a + b;
        return result;
    }
}
```
USING THE INTERFACE

- take the interface of another contract to know the functions that can be invoked
- to invoke the functions:
  - get an instance of the contract implementing the interface by passing the address to the instance

```
pragma solidity ^0.6.0;
interface A {
    function f1(bool arg1, uint arg2) external returns (uint);
}
contract YourContract {
    function doYourThing(address AddressOfA) public returns (uint) {
        A myA = A(AddressOfA);
        return myA.f1(true, 3);
    }
}
```
pragma solidity ^0.6.0;

contract Token{
        mapping (address => uint) public balances;
    constructor () public {
            balances[msg.sender]= 1000000; }
    modifier only_with_at_least (uint x){
            require(balances[msg.sender] >= x); _ ;}
    function transfer(uint amount, address dest) public only_with_at_least(100){
            balances[msg.sender]-= _amount;
            balances[_dest]+= _amount;
    }}

• in the modifier body: _ (underscore) represents function body
• only_with_at_least: abstracting the notion of “executing account must have a balance of at least some particular amount”
• avoid to mix pre-condition logic with state-transition logic
• reuse the same modifier in different contexts
A FACTORY CONTRACT

- imagine to define a simple contract which encapsulates an integer counter
  - a function to increase the counter
  - a function to query the value of the counter
- goal: to define a factory of Counter contracts that
  - creates a Counter contracts on behalf of external entities requiring it
  - invokes the functions of the Counter contract of behalf of the owner of the contract
  - use modifiers in the contracts definition
pragma solidity 0.6.0;

contract Counter {
    uint256 private _count;
    address private _owner;
    address private _factory;

    modifier onlyOwner(address caller) {
        require(caller == _owner, "You're not the owner of the contract");_; }
    modifier onlyFactory() {
        require(msg.sender == _factory, "You need to use the factory");_;
    }

    constructor(address owner) public {
        _owner = owner;
        _factory = msg.sender;
    }

    function getCount() public view returns (uint256) {
        return _count;
    }

    function increment(address caller) public onlyFactory onlyOwner(caller) {
        _count++;
    }
}
contract CounterFactory {
    mapping(address => Counter) _counters;

    function createCounter() public {
        require (_counters[msg.sender] == Counter(0));
        _counters[msg.sender] = new Counter(msg.sender);
    }

    function increment() public {
        require (_counters[msg.sender] != Counter(0));
        Counter(_counters[msg.sender]).increment(msg.sender);
    }

    function getCount(address account) public view returns (uint256) {
        require (_counters[account] != Counter(0));
        return (_counters[account].getCount());
    }

    function getMyCount() public view returns (uint256) {
        return (getCount(msg.sender));
    }
}