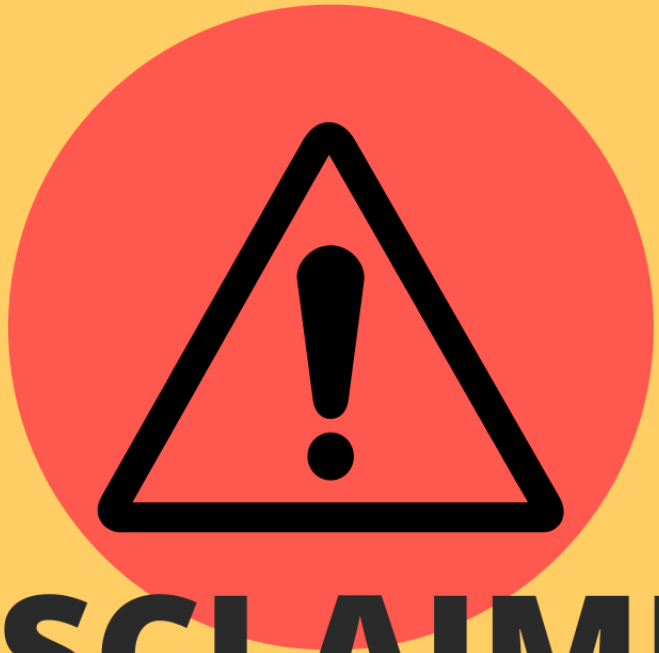

DISCLAIMER

Il materiale contenuto nel drive è stato raccolto e richiesto tramite autorizzazione ai ragazzi frequentanti il corso di studi di Informatica dell'Università degli Studi di Salerno. Gli appunti e gli esercizi nascono da un uso e consumo degli autori che li hanno creati e risistemati per tanto non ci assumiamo la responsabilità di eventuali mancanze o difetti all'interno del materiale pubblicato.

Il materiale sarà modificato aggiungendo il logo dell'associazione, in tal caso questo possa recare problemi ad alcuni autori di materiale pubblicato, tale persona può contattarci in privato ed elimineremo o modificheremo il materiale in base alle sue preferenze.

Ringraziamo eventuali segnalazioni di errori così da poter modificare e fornire il miglior materiale possibile a supporto degli studenti.

CoScienze
Associazione

DESCRIZIONE SMART CONTRACT SOLIDITY

———Descrivere cosa realizza il seguente smart contract in Solidity:

```
pragma solidity >0.4.23 <0.6.0;
contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }
    address payable public beneficiary;
    uint public biddingEnd;
    uint public revealEnd;
    bool public ended;

    mapping(address => Bid[]) public bids;
    address public highestBidder;
    uint public highestBid;
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    modifier onlyBefore(uint _time) { require(now < _time); _; }
    modifier onlyAfter(uint _time) { require(now > _time); _; }

    constructor(
        uint _biddingTime,
        uint _revealTime,
        address payable _beneficiary
    ) public {
        beneficiary = _beneficiary;
        biddingEnd = now + _biddingTime;
        revealEnd = biddingEnd + _revealTime;
    }

    function bid(bytes32 _blindedBid)public payable
        onlyBefore(biddingEnd)
    {
        bids[msg.sender].push(Bid({
            blindedBid: _blindedBid,
            deposit: msg.value
        }));
    }
    function reveal(
        uint[] memory _values,
        bool[] memory _fake,
        bytes32[] memory _secret
    )public
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
        uint length = bids[msg.sender].length;
        require(_values.length == length);
        require(_fake.length == length);
        require(_secret.length == length);

        uint refund;
        for (uint i = 0; i < length; i++) {
            Bid storage bidToCheck = bids[msg.sender][i];
            (uint value, bool fake, bytes32 secret) =
                    (_values[i], _fake[i], _secret[i]);
```

CoScienze Associazione

```
            if (bidToCheck.blindedBid != keccak256(abi.encodePacked(value, fake,
secret))) {
                continue;
            }
            refund += bidToCheck.deposit;
            if (!fake && bidToCheck.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
            }
            bidToCheck.blindedBid = bytes32(0);
        }
        msg.sender.transfer(refund);
    }

    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != address(0)) {
            // Refund the previously highest bidder.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
    }

    function withdraw() public {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            pendingReturns[msg.sender] = 0;

            msg.sender.transfer(amount);
        }
    }

    function auctionEnd()
        public
        onlyAfter(revealEnd)
    {
        require(!ended);
        emit AuctionEnded(highestBidder, highestBid);
        ended = true;
        beneficiary.transfer(highestBid);
    }
}
```

CoScienze Associazione

Il seguente contratto è un contratto di aste, dove i partecipanti inviano offerte alla cieca e vince chi fa la puntata più alta. Durante il periodo di offerta , l'offerente non invia effettivamente la propria offerta, ma solo una versione con hash e si impegna all'offerta con ciò. Dopo la fine del periodo di offerta, gli offerenti devono rivelare le loro offerte: inviano i loro valori in chiaro e il contratto verifica che il valore hash sia lo stesso di quello fornito durante il periodo di offerta.
Il contratto accetta qualsiasi valore maggiore dell'offerta più alta. Poiché questo può ovviamente essere verificato solo durante la fase di rivelazione, alcune offerte potrebbero non essere valide.

- Pragma indica la versione con cui verrà compilato lo smart contract dalla EVM
- C'è la creazione del contratto BlindAuction con struttura dati Bid avente come parametri una variabile blindeBid di tipo bytes32 e una variabile deposito di tipo unisigned int.

Abbiamo la dichiarazione delle variabili necessarie al funzionamento del contratto, tra cui beneficiary che è una variabile pubblica di tipo payable, ovvero accetta transazioni.
Vi sono due tabelle hash chiave/valore in cui nella prima viene assegnato agli address le Bid[ ], rappresentata da bids e per la seconda un associazione agli address valori di tipo uint, rappresentata da pendingReturns.

- Viene specificato poi un evento AuctionEnded che prende in input l'indirizzo del winner e la puntata più alta.
- I modifier sono un modo conveniente per convalidare gli input alle funzioni, vi è poi il costruttore di tipo pubblico dove vengono inizializzate le variabili e delle relative assegnazioni.
- La funzione Bid di tipo payable riguarda una transazione, ossia l'invio di una bid con i relativi dati e inoltre c'è la verifica temporale. ossia che il tempo attuale è inferiore al tempo relativo alle fine del tempo delle puntate (biddingEnd).
- La funzione reveal va a svelare le offerte alla cieca ed effettua un controllo su tutte le puntate ricevute, Al termine del controllo riceveranno un rimborso tutte le offerte blindedBid non valide e tutte le altre offerte tranne quella highestBid.
- La funzione placeBid è una funzione "interna", il che significa che può essere chiamata solo dal contratto stesso (o dai contratti derivati). In questa funzione abbiamo il riferimento al puntatore e al valore offerto, c'è un controllo sulla puntata e quella più alta, se la puntata più alta ha indirizzo differente dal puntatore dell'offerta allora ci sarà il rimborso dell'offerta al puntatore, altrimenti se il puntatore ha fatto l'offerta più alta si pone l'offerta più alta uguale all'offerta del puntatore.
- C'è poi la funzione withdraw di tipo pubblico , per il prelievo e ritira un'offerta che è stata superata.
- Infine la funzione auctionEnd di tipo pubblico , dove termina l'asta e invia l'offerta più alta beneficiario

—----------

Descrivere cosa realizza il seguente smart contract in solidity

```
pragma solidity ^0.4.22;
contract SimpleAuction {
    address public beneficiary;
    uint public auctionEnd;
    address public highestBidder;
    uint public highestBid;
    mapping(address => uint) pendingReturns;
    bool ended;
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    constructor(
        uint _biddingTime,
        address _beneficiary
    ) public {
        beneficiary = _beneficiary;
        auctionEnd = now + _biddingTime;
    }
    function bid() public payable {
        require(
            now <= auctionEnd,
            "Auction already ended."
        );
        require(
            msg.value > highestBid,
            "There already is a higher bid."
        );
        if (highestBid != 0) {
            pendingReturns[highestBidder] += highestBid;
```

```
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        emit HighestBidIncreased(msg.sender, msg.value);
    }
    function withdraw() public returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            pendingReturns[msg.sender] = 0;
            if (!msg.sender.send(amount)) {
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }
    function auctionEnd() public {
        require(now >= auctionEnd, "Auction not yet ended.");
        require(!ended, "auctionEnd has already been called.");
        ended = true;
        emit AuctionEnded(highestBidder, highestBid);
        beneficiary.transfer(highestBid);
    }
}
```

RISPOSTA

Questo smart contract in Solidity sostanzialmente rappresenta delle transazioni per mezzo di offerte , intese come puntate. Vengono dunque selezionate le puntate maggiori per effettuare i vari controlli. Simple action è il nome del contratto. vi è poi l'inizializzazione delle variabili beneficiary e auctionendtime, rispettivamente di tipo indirizzo e intero senza segno, entrambe pubbliche, inoltre la prima è anche di tipo payable dunque utilizzabile per l'esecuzione di una transazione. sono inizializzate poi highestBidder ed highestBid, i tipo intero senza segno ed indirizzo. Vi è poi una struttura dati dinamica, chiave/valore, in cui degli interi senza segno vengono assegnati agli address, essa rappresenta pendingReturn.
Inizializzo una variabile booleana ended.
Vengono poi specificati due eventi , event highestBidIncresead e AuctionEnded. Prendono in input rispettivamente l'indirizzo del binder ed una cifra amount. Mentre l'altra l'indirizzo dell' winner e la cifra da aumentare. Vi è poi il costruttore di tipo pubblico, in cui vengono inizializzate delle variabili e delle relative assegnazioni. - La funzione Bid di tipo payable, riguardante dunque una transazione, verifica che il limite temporale attuale è minore uguale ad auctionendtime. Viene poi verificato che il valore di chi chiama, sia maggiore di highestbid. Nel caso in cui highestbid è inoltre diverso da 0, viene eseguita l'operazione con assegnamento. viene assegnato colui che ha deployato il contratto ad highestbidder ed il valore ad highestbid. Con emit, viene dunque emesso l'evento specificato, ovvero highestbidincreased.
C'è poi la funzione withdraw, per il prelievo. Essa è di tipo publica e ritorna un valore booleano. il valore pendente di chi ha deployato, viene assegnato a amount. Se questo amount è maggiore di 0, esegue l'istruzione di assegnamento. Se msg.sender.amount non è vera, allora assegna amount come pendingRetourn di chi ha deployato e poi ritorna falso. se l'amount iniziale era minore di 0, ritorna true. La funzione auctioneEnd è di tipo pubblica, appunto serve a terminare l'asta. Con il primo require si verifica che l'istante attuale è maggiore o uguale a quello di fine dell' auction. Viene poi verificato se not ended è true, nel caso, ended viene posto a true. Viene poi emesso l'evento auctionEnded, che prende in input il bidder più altro e l'highestbid. Infine, dunque il beneficiario trasferisce il valore highestBid.

———Descrivere cosa realizza il seguente smart contract in Solidity: //21/12/2021

```
pragma solidity ^0.8.10;
import "https://github.com/OpenZeppelin/openzeppelin-
contracts/blob/v4.0.0/contracts/token/ERC20/IERC20.sol";

contract TokenSwap {
```

```
    IERC20 public token1;
    address public owner1;
    uint public amount1;
    IERC20 public token2;
    address public owner2;
    uint public amount2;

    constructor(
        address _token1,
        address _owner1,
        uint _amount1,
        address _token2,
        address _owner2,
        uint _amount2
    ) {
        token1 = IERC20(_token1);
        owner1 = _owner1;
        amount1 = _amount1;
        token2 = IERC20(_token2);
        owner2 = _owner2;
        amount2 = _amount2;
    }

    function swap() public {
        require(msg.sender == owner1 || msg.sender == owner2, "Not authorized");
        require(
            token1.allowance(owner1, address(this)) >= amount1,
            "Token 1 allowance too low"
        );
        require(
            token2.allowance(owner2, address(this)) >= amount2,
            "Token 2 allowance too low"
        );

        _safeTransferFrom(token1, owner1, owner2, amount1);
        _safeTransferFrom(token2, owner2, owner1, amount2);
    }

    function _safeTransferFrom(
        IERC20 token,
        address sender,
        address recipient,
        uint amount
    ) private {
        bool sent = token.transferFrom(sender, recipient, amount);
        require(sent, "Token transfer failed");
    }
}
```

RISPOSTA

Questo contratto sostanzialmente effettua lo scambio di token tra due indirizzi.

Pragma indica la versione con cui la EVM compilerà lo smart contract.

Successivamente viene importata la libreria Openzeppelin importata da Github.

TokenSwap indica il nome del contratto e al sui interno vengono definite le variabili pubbliche dei due soggetti dello scambio.

Viene inizializzato un costruttore che al suo interno prende tutte le variabili definite in precedenza e inizializza le variabili del contratto.

La funzione Swap è una funzione pubblica, quindi può essere acceduta da altri contratti. Questa funzione effettua un controllo su chi invoca il contratto e se

CoScienze Associazione

è uno dei due soggetti dello scambio a chiamare il contratto allora restituisce un messaggio di errore. Subito dopo vengono effettuati 2 controlli che verificano se gli amount sono minori dell'allowance, in tal caso i controlli restituiranno un messaggio di errore. Nel caso in cui tutti i controlli vengano superati con successo, viene invocata la funzione safeTransferFrom che prende in input il Token1, l'indirizzo del sender, il destinatario e l'amount e la stessa cosa per il per il token2 .

Infine abbiamo la funzione safeTranseferFrom che è privata, questo significa che può essere invocata solo dal contratto. Il contratto quando invoca la funzione trasferisce il bene da un indirizzo all'altro e restituisce un valore booleano, nel caso in cui la transazione fallisce viene stampato un messaggio di errore

---

Descrivere cosa realizza il seguente smart contract in Solidity:

```
pragma solidity >=0.4.24 <0.6.0;

contract ReceiverPays {
   address owner = msg. sender ;
   mapping ( uint256 => bool ) usedNonces ;

   constructor () public payable {}

   function claimPayment ( uint256 amount , uint256 nonce , bytes memory
signature )
        public {
        require (! usedNonces [ nonce ]);
        usedNonces [ nonce ] = true ;
        bytes32 message = prefixed ( keccak256 ( abi. encodePacked (msg .sender
, amount , nonce , this )));
        require ( recoverSigner ( message , signature ) == owner );
        msg . sender . transfer ( amount );
  }

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

  function splitSignature ( bytes memory sig )
   internal
   pure
   returns ( uint8 v, bytes32 r, bytes32 s)
  {
   require ( sig . length == 65) ;
   r := mload (add (sig , 32) )
   s := mload (add (sig , 64) )
   v := byte (0, mload ( add (sig , 96) ))
   return (v, r, s);
  }

  function recoverSigner ( bytes32 message , bytes memory sig )
   internal
   pure
   returns ( address )
  {
   ( uint8 v, bytes32 r, bytes32 s) = splitSignature (sig );
   return ecrecover ( message , v, r, s);
  }
```

CoScienze Associazione

```
  function prefixed ( bytes32 hash ) internal pure returns ( bytes32 ) {
    return keccak256 ( abi . encodePacked ("\ x19Ethereum Signed Message :\ n32
", hash ));
  }
}
```

RISPOSTA:

Questo smart contract effettua una transazione, ovvero un pagamento, e con una serie di controlli sulle stesse, verifica l'attendibilità e cioè la firma. Infine, se tutto viene verificato, restituisce l'hashcode della transazione.

Pragma Solidity rappresenta la versione con cui AWM compila il contratto.

Abbiamo il mapping con chiave/valore, dove la chiave è uint256 e il valore è di tipo booleano, riferendosi a usedNonces.

Abbiamo il costruttore di topo payable vuoto.

La funzione claimPayment, prende in input amout, nonce e memory signature, di tipo pubblica, questo significa che può essere acceduta da altri contratti, verifica se usedNonce non è vera, in questo caso fallisce, nel caso sia vera lo assegna a true.

Viene fatto l'hash del messaggio e si va a controllare che all'owner corrispondano il messaggio e la firma.

Nel caso viene superato questo controllo la transazione viene eseguita correttamente.

La funzione kill di tipo pubblico, è una funzione di autodistruzione che si attiva quando l'owner corrisponde a colui che ha deployato lo smart contract (sender)

La funzione splitSignature, di tipo interno, cioè può essere acceduta sia dal contratto che dai suoi derivati, è una funzione che prende in input memory sig. Essa verifica se la lunghezza della firma è uguale a 65, se passa questo controllo effettua le operazioni nel corpo e restituisce i valori corrispondenti alle variabili

La funzione recoverSigner interna prende in input, il messaggio e il memory sig, ritorna l'address se v,r,s corrispondono al ritorno della funzione splitSignature, in address abbiamo il messaggio, v, r ed s.

La funzione prefixed prende in input un hash, di tipo interni e ritorna un bytes 32, ritornando la codifica di Ehterium indicando con un messaggio l'hash della transazione.

—----------

```
pragma solidity ^0.4.22;
contract Ballot {
struct Voter {
        uint weight;
        bool voted;
        address delegate;
        uint vote;
 }
struct Proposal {
        bytes32 name;
        uint voteCount;
 }
```

```
 address public chairperson;
 mapping(address => Voter) public voters;
 Proposal[] public proposals;

constructor(bytes32[] proposalNames) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;
        for (uint i = 0; i < proposalNames.length; i++) {
                proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
                }));
        }
 }

function giveRightToVote(address voter) public {
        require(msg.sender == chairperson,"Only chairperson can give right to
vote.");
        require(!voters[voter].voted,"The voter already voted.");
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
 }

function delegate(address to) public {
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "You already voted.");
        require(to != msg.sender, "Self-delegation is disallowed.");
        while (voters[to].delegate != address(0)) {
                to = voters[to].delegate;
                require(to != msg.sender, "Found loop in delegation.");
 }
 sender.voted = true;
 sender.delegate = to;
 Voter storage delegate_ = voters[to];
 if (delegate_.voted) {
        proposals[delegate_.vote].voteCount += sender.weight;
 } else {
        delegate_.weight += sender.weight;
  }
 }


function vote(uint proposal) public {
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;
        proposals[proposal].voteCount += sender.weight;
 }

function winningProposal() public view returns (uint winningProposal_){
         uint winningVoteCount = 0;
         for (uint p = 0; p < proposals.length; p++) {
                if (proposals[p].voteCount > winningVoteCount) {
                        winningVoteCount = proposals[p].voteCount;
                        winningProposal_ = p;
                }
         }
}

function winnerName() public view returns (bytes32 winnerName_){
        winnerName_ = proposals[winningProposal()].name;
        }
```

CoScienze Associazione

}

RISPOSTA:

Pragma Solidity rappresenta la versione con cui EVM compila il contratto.

Abbiamo una struttura Voter, contenente delle variabili relative al votante.
Abbiamo una struttura proposal contenente il nome e il numero di voti ricevuti.

Abbiamo chairperson, pubblico di tipo address poi abbiamo il mapping chiave/valore dei votanti, dove la chiave è l'address e il valore è il Voter.

Abbiamo una lista pubblica di proposal.

Un costruttore pubblico che prende in input il proposalNames, che crea una nuova scheda per scegliere uno dei `ProposalNames`.
Per ciascuno dei ProposalNames forniti, crea un nuovo oggetto proposta e aggiungilo alla fine dell'array.

La funzione pubblica giveRightToVote prende in input un voter, e controlla se il sender è uguale al chairperson, solo il chariperson può dare al voter " il diritto di votare su questa scheda.
Se il primo argomento di `require` restituisce `false`, l'esecuzione termina e tutte le modifiche allo stato e ai saldi Ether vengono annullate.

La funzione delegate di tipo pubblico prende in input l'address di un destinatario "to", serve a delegare il voto all'elettore `to`. Controlla se "to" ha già votato, e controlla se il sender è diverso dal destinatario "to" dato che l'auto delega non è consentita. Nel while inoltra la delega purchè "to" sia delegato e controlla se il delegato "to" è diverso dal sender, in caso siano uguali il while va in loop, in caso contrario si va avanti e viene effettutato un controllo sul delegato ha già votato, quindi aggiunge direttamente al numero di voti, altrimenti se il delegato non ha già votato aggiungi al suo weight.

La funzione vote, pubblica prende in input un proposal, controlla se il sender ha già votato , se non ha votato allora aumento il conuter dei voti con il weight del sender.

La funzione winningProposal, pubblica cancola la proposta vincente tenendo conto di tutti i voti precedenti
La funzione winningName pubblica, chiama la funzione winningProposal, e ritorna il nome del vincitore

RISPOSTA_ANDREA:

Il seguente smart contrac, di nome Ballot, banalmente constente di effettuare correttamente delle votazioni, andando nello specifico:

Pragma solodity indica la versione con cui l'EVM andrà a compilare lo smart contract.

Nel contratto Ballot, abbiamo:

La prima struttura di tipo Voter, dove è definito uint weight, ossia il penso di tipo usigned int, boolean voted, quindi se già si ha votato o meno , address delegate, quindi l'indirizzo di chi è delegato a votare ed infine uint vote, quindi anche il voto è ti tipo usigned int.

Abbiamo la seconda struttura di tipo Proposal, devo abbiamo, bytes32 name, quindi il nome del proposal in byte, e uint voteCount, quindi il contatore unsigned int, dei voti associati al nome del proposal.

Abbiamo l'address del chairperson, quindi del direttore delle votazioni.

Abbiamo il mapping in voters, ossia una map, quindi una memoria associativa, dove nell'array vengono salvate le coppie chiave/valore, la chiave in questo è l'indirizzo , address, e il valore associato è la struttura Voter.

Nella variabile proposal abbiamo il riferimento alla struttura Proposal.

CoScienze Associazione

Il costruttore prende in input un array di byte32, chiamato proposalName, abbiamo che nel chairpersone viene messo il message sender, e nella memoria associata tutti i voters di tipo chairpersone hanno weight 1, quindi peso 1. Poi c'è un for che ha la durata uguale al valore della lunghezza dell'array preso in input, dove nella lista di proposal viene caricato il il nome di proposalName[i] , quindi l'elemento associato all'indice del ciclo for, e con il contatore dei voti voteCount a 0 .

Nella funzione giveRightVote, che prende in input gli indirizzi dei voters (la memoria associativa), abbiamo il controllo che il sender sia uguale al chairperson, dato che solo i presidenti possono dare il diritto al voto. C'è il controllo che il votante non abbia già votato ,altrimenti c'è l'errore il votante ha già votato. C'è il controllo che il weight, il peso del votante sia uguale a 0, e se è cosi imposta il weight a 1.

Poi c'è la funzione delegate , che prende in input un address to (quindi l'indirizzo del delegato "to") , c'è la creazione di una seconda struttura di Voter avente nome storage sender, che viene riempita con le informazioni del sender contenuta nella memoria associativa voter. Controlla se il delegante abbia già votato, se ha già votato, c'è l'errore hai già votato, altrimenti, c'è il controllo se il delegante è uguale al delegato, se è uguale, c'è l'errore che l'auto delegazione non è consentita, se passa il controllo c'è il ciclo while, abbiamo la condizione che nella memoria associata voters, il delegato "to" non abbia già votato e il controllo che il delegato non sia il delegante, se non sono uguali il while non va il loop e si può continuare,

si imposta il delegante con il voted a true , quindi il delegante ha votato e si imposta l'indirizzo con il delegato "to" e si aggiunge alla struttura di tipo Voter, storage delegate , il votante "to" . Se il delegato ha quindi votato, true, si passa l'if e si aggiunge alla struttura proposal di tipo Proposer il numero dei voti, dato in base al peso, se tutto è andato correttamente si aggiunge il peso del delegante 1 , altrimenti de il delegato non ha votato si aggiunge al peso del delegato il peso del delegante (0 + 1).

Nella funzione vote, che prende in input uint proposal, abbiamo , la stuttura storare sender di tipo Voter dove vengono inseriti i votanti della memoria ausiliaria voters, si controlla se il votante ha già votato, se ha votato c'è l'errore già hai votato, altirmenti si imposta al votante , voted a true, quindi ha votato, e si vanno ad aggiungere i voti, a countVote, alla stuttura proposals di tipo Proposal.

Nella funzione winningProposal, abbiamo il ritorno di uint, unisgned int, winningProposal_ , dove abbiamo, che la variabile WinningVoteCount è uguale a 0, abbiamo un for della durata uguale al valore della lunghezza della lista di proposals, dove, se il proposal, proposto, nella posizione i-esima ha un numero di voti maggiore a WinningVoteCount, allora si va a mettere nella variabile winningProposal_ il proposal p associato al numero più alto di voti, la funzione quindi ritorna il proposer p , con il numero di voti più alto

Infine la funzione winnerName, ritorna banalmente il nome del vincitore, nella variabile winnerName_, al suo interno abbiamo che sulla variabile winnerName_ viene chiamata la funzione winnigProposal.name, che restituirà il nome del vincitore del ballottaggio.

CoScienze Associazione

Quindi lo smart contract in questione permette ai votanti selezionati dal presidente del ballotaggio, di votare una volta, i votanti possono avere un delegato al voto, tutti i voti validi, hanno peso uguale ad 1, i voti vengono aggiunti ad un contatore associato ad ogni singolo proposal, e il proposal con il numero di voti maggiore vince il ballottaggio, e quindi alla fine viene restituito il nome del vincitore.

DOMANDA 5

```
contract MyToken {

    string public name;
    string public symbol;
    uint8 public decimals;

    mapping (address => uint256) public balanceOf;

    event Transfer(address indexed from, address indexed to, uint256 value);

    function transfer(address _to, uint256 _value) public {
        require(balanceOf[msg.sender] >= _value && balanceOf[_to] + _value >=
balanceOf[_to]);
        balanceOf[msg.sender] -= _value;
        balanceOf[_to] += _value;
```

```
        emit Transfer(msg.sender, _to, _value);
    }

   constructor(uint256 initialSupply, string memory tokenName, string memory
tokenSymbol, uint8 decimalUnits) public {
        balanceOf[msg.sender] = initialSupply;
        name = tokenName;
        symbol = tokenSymbol;
        decimals = decimalUnits;
    }
}
```

RISPOSTA

Il contratto MyToken trasferisce una quantità di token da un indirizzo ad un altro. Abbiamo la dichiarazione delle variabili , nome simbolo e decimali e poi fa il mapping chiave/valore per balanceOf.

Abbiamo l'evento di trasferimento da un indirizzo from all'indirizzo to, con un determinato valore di token da trasferire, che verrà attivato alla ricezione del segnale emit.

La funzione Tranfer pubblica, prende in input un indirizzo to e un valore, c'è il requisito che il saldo del sender sia maggiore o uguale al valore da trasferire AND la somma del saldo del destinatario e del valore sia maggiore o uguale al saldo del destinatario iniziale.

Se la richiesta è soddisfatta va a sottrarre al saldo del sender il valore inviato e lo incrementa a quello del destinatario.

Con emit, viene dunque emesso l'evento specificato, ovvero Transfer

Poi il costruttore ha come parametri il saldo iniziale, la memoria del nome del token, la memoria del simbolo del token e le unità decimali, e va a dichiarare ed inizializzare le variabili corrispondenti.

CoScienze Associazione

```
pragma solidity >=0.4.11 <0.6.0;

contract CrowdFunding {
    struct Funder {
        address addr;
        uint amount;
    }

    struct Campaign {
        address payable beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping (uint => Funder) funders;
    }

    uint numCampaigns;
    mapping (uint => Campaign) campaigns;

    function newCampaign(address payable beneficiary, uint goal) public returns
(uint campaignID) {
        campaignID = numCampaigns++; // campaignID is return variable
```

```
        campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
    }

    function contribute(uint campaignID) public payable {
        c.funders[c.numFunders++] = Funder({addr: msg.sender, amount:
msg.value});
        c.amount += msg.value;
    }

    function checkGoalReached(uint campaignID) public returns (bool reached) {
        Campaign storage c = campaigns[campaignID];
        if (c.amount < c.fundingGoal)
            return false;
        uint amount = c.amount;
        c.amount = 0;
        c.beneficiary.transfer(amount);
        return true;
    }
}
```

RISPOSTA

Pragma indica la versione con cui la EVM compila il contratto. Il contratto si chiama Crowfounding e la struttura Funder (finanziamento) dichiara 2 variabili: address e amount che è di tipo int senza segno.
Poi abbiamo la struttura Campaign dove al suo interno vi sono le variabili: beneficiario di tipo payable, ovvero che può ricevere transazioni; il foundinggoal, il numero di founders, l'amount ed in fine il mapping, ovvero la coppia chiave/valore dove la chiave è di tipo uint e il valore sono i Founder, il tutto definito in founders. Successivamente viene definita una variabile contatore per il numero di Campaign ed il mapping, ovvero la coppia chiave/valore dove la chiave è di tipo uint e il valore sono le Campaign, il tutto definito in campaigns.
La funzione newCampaing prende in input l'indirizzo del beneficiario e un obiettivo, questa funzione è di tipo pubblico, ovvero può può essere acceduto da altri contratti e ritorna il campaignID. All'interno della funzione viene incrementato il contatore delle campagne e viene riempita la struttura della campaigns con quella associata.
Poi c'è la funzione contribute che prende in input il campaignID ed è pubblica e payable, ovvero può essere acceduta da altri contratti e può ricevere transazioni. In questa funzione vengono aggiunti tutti i founders e viene incrementato il valore dell'amount.
Infine abbiamo la funzione checkGoalReached che prende in input il campaignID, è di tipo pubblica e restituisce un booleano reached per indicare se il goal sia stato raggiunto. In questa funzione abbiamo che se l'amount è minore dell'obiettivo ritorna false, altrimenti true ed il conseguente trasferimento del valore al beneficiario.

DOMANDA 2

```
contract Purchase {
    uint public value;
    address public seller;
    address public buyer;
    enum State { Created, Locked, Inactive }
    State public state;
```

```
constructor() public payable {
    seller = msg.sender;
    value = msg.value / 2;
    require((2 * value) == msg.value, "Value has to be even.");
}

modifier condition(bool _condition) {
    require(_condition);
    _;
}

modifier onlyBuyer() {
    require(
        msg.sender == buyer,
        "Only buyer can call this."
    );
    _;
}

modifier onlySeller() {
    require(
        msg.sender == seller,
        "Only seller can call this."
    );
    _;
}

modifier inState(State _state) {
    require(
        state == _state,
        "Invalid state."
    );
    _;
}

event Aborted();
event PurchaseConfirmed();
event ItemReceived();

function abort()
    public
    onlySeller
    inState(State.Created)
{
    emit Aborted();
    state = State.Inactive;
    seller.transfer(address(this).balance);
}

function confirmPurchase()
    public
    inState(State.Created)
    condition(msg.value == (2 * value))
    payable
{
    emit PurchaseConfirmed();
    buyer = msg.sender;
    state = State.Locked;
}

function confirmReceived()
    public
```

CoScienze Associazione

```
        onlyBuyer
        inState(State.Locked)
    {
        emit ItemReceived();
        // It is important to change the state first because
        // otherwise, the contracts called using `send` below
        // can call in again here.
        state = State.Inactive;

        // NOTE: This actually allows both the buyer and the seller to
        // block the refund - the withdraw pattern should be used.

        buyer.transfer(value);
        seller.transfer(address(this).balance);
    }
}
```

RISPOSTA

Questo codice si occupa, come dice il titolo, di un ordine, in questo caso si trattaa dell'acquisto di un bene.

Pragma indica la versione con cui il contratto viene compilato dall'AWM.

Purchase indica il nome del contratto e al suo interno vengono definite una variabile pubblica senza segno value una variabile seller e buyer di tipo address e pubblici. l'enum dello stato delle variabili e uno state pubblico.

Il costruttore che è pubblico e di tipo payable,payable significa che si possono ricevere le transazioni, va a verificare che il value del seller sia un numero pari.

I modifier sono un modo conveniente per convalidare gli input alle funzioni.
Poi vengono definiti tre eventi Aborted(), PurchaseConfirmed() e ItemReceived().

La funzione Abort, pubblica, può essere chiamata solo dal seller prima che il contratto sia bloccato e recupera l'ether.

La funzione confirmPurchase, conferma l'acquisto come acquirente è di tipo payable ovvero può ricevere transazioni e deve ricevere valori pari e blocca l'ether finchè non verrà chiamata la funzione PurchaseConfirmed

La funzione cofirmReiceved, è pubblica, e può essere chiamata solo dal compratore, questa funzione conferma che l'acquirente ha ricevuto l'oggetto e sblocca l'ether

---

```
pragma solidity ^0.4.18;

contract Oracle {
    struct Request {
    bytes data;
    function(bytes memory) external callback;
    }
    Request[] requests;
    event NewRequest(uint);
    function query(bytes data, function(bytes memory) external callback) {
    requests.push(Request(data, callback));
```

```
    NewRequest(requests.length - 1);
    }
    function reply(uint requestID, bytes response) {
    // Here goes the check that the reply comes from a trusted source
    requests[requestID].callback(response);
    }
}

contract OracleUser {
    Oracle constant oracle = Oracle(0x1234567); // known contract
    function buySomething() {
    oracle.query("USD", this.oracleResponse);
    }
    function oracleResponse(bytes response) {
    require(msg.sender == address(oracle));
    }

}
```

RISPOSTA

Il contratto Oracle

CoScienze
Associazione