P2P Systems and Blockchains

Spring 2020,
instructor: Laura Ricci
laura.ricci@unipi.it

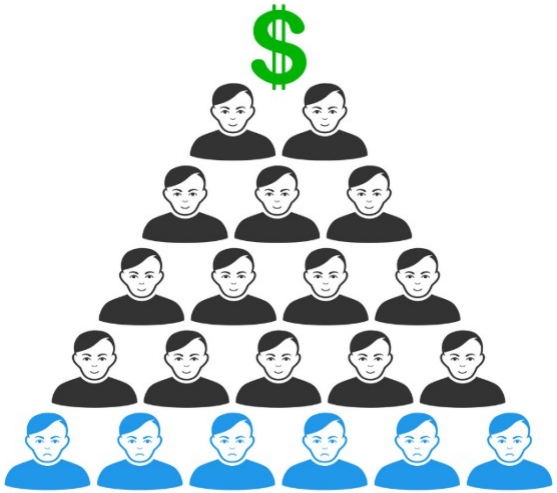
Lesson 16:
SMART CONTRACTS: SOLIDITY AND REMIX

24/04/2020

PROGRAMMING SMART CONTRACTS: SOLIDITY

- a contract oriented high level language for EVM
- a static type system
- a syntax very similar to javascript
- a language in continuos evolution
  - last version v0.6.6
  - https://solidity.readthedocs.io/en/v0.6.6/
- 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

the first Solidity program: a simplified Ponzi Scheme

- for each investment, take the money and send it to the previous investor
- each investment must be larger than the previous one
- each investor except the last one will get a return on their investment

A PONZI SCHEME IN SOLIDITY

```
pragma solidity ^0.6.0;
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

```
pragma solidity ^0.5.2;
```

define the allowed range of versions of the compiler

- useful for languages in evolution, like Solidity
  - matches version of the compiler ≥ 0.5.0 and < 0.6.0

- includes a major build (5), followed by a minor build number
  - the caret (^) before the version number points out that
    - Solidity can use the latest build in the major version range, but not later versions
    - future compiler versions that might introduce incompatible changes
  - no caret: all the version newer than the one referred
  - more complex expressions can be specified to define versions ranges

A SIMPLE PONZI SCHEME IN SOLIDITY

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

`currentInvestment`

- the amount of the investment lost if there are no further investments

`currentInvestor` address of the most recent investor in the contract.

- the only one that has not yet received a return on the investment
- the one which will lose his/her investment if no one make an investment

- contract is like a class

THE ADDRESS VALUE TYPE

```
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 = 0xE0f5206BBD039e7b0592d918820024e2a7437b9;
//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)
```

SOLIDITY DATA TYPES

```
contract SimplePonzi {
    address payable currentInvestor;
    uint public currentInvestment; bool notfirst;
```

- integer types
  - `int` (signed)
  - `uint` (unsigned)
- keywords: `uint8` / `int8` to `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).
- no floats!

SOLIDITY DATA TYPES

type	Default value
`uint8-uint256`	`0` (zero)
`int8-int256`	`0` (zero)
`bool`	`false`
`string`	An empty string
`byte`	In integer form, `0`; in hex form, `0x00`
`bytes1-bytes32`	In integer form, `0`; in hex form, `0x<NumberOfZeros>` (*NumberOfZeros* = *NumberOfBytes* * 2; for example, for `bytes2`, the default value is `0x0000`)
`bytes`	Empty bytes prefixed with `0x`
`address`	`0x0000000000000000000000000000000000000000`

A default value is defined for each type

FALLBACK FUNCTIONS

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

- 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 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

PAYABLE FUNCTIONS

```
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 parameteres
  - adding a value is like putting cash inside the envelope
- a `payable` function
  - requires a payment to execute
  - 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

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.*

REQUIRE: TESTING CONDITIONS

```
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 variables and functions existing in the global namespaceprovide information on the transactions

- 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.sig function identifier (bytes4)
  - msg.data complete calldata (bytes)

- tx.gasprice gas price of the transaction

- tx.origin original sender of the transaction

SENDING VALUES

```
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 new next investment
- three ways to send ether in Solidity:
  - address.transfer(value)
  - address.send(value)
  - address.call.value(value)()

TRANSFERRING VALUES

- address.transfer (value)
  - transfesr 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.

A MORE REALISTIC PONZI

```
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();}}
```

FIXED AND DYNAMIC SIZED ARRAYS

```
pragma 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
  - lenght 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 implicitily boind to all-zero value
  - binding is always defined

ETHER UNITS

```
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

Suffix example	In wei
1 wei	1
1 szabo	1e12
1 finney	1e15
1 ether	1e18

A LOTTERY ON THE BLOCKCHAIN

- run a lottery on the blockchain, without anchoring its operation in any single legal jurisdiction
- need a source of entropy in a deterministic environment. Possible solutions:
  - use an external oracle to minimize complexity and external dependencies
    - does not minimize complexity and external dependencies
  - use the data on the blockchain, for instance the blockhash, as a source of randomness
    - miners could cheat and influence a value
  - more complex solutions
    - RanDAO
    - Publicly Veriafiable Secret Sharing
    - Verifiable Delay Functions, Verifiable Random Functions
- our solution; use data on the blockchain

A LOTTERY ON THE BLOCKCHAIN

```
pragma solidity ^0.6.0;

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

    constructor (uint duration) public {
        ticketingCloses = now + duration;   }

    function buy () public payable {
        require(msg.value == TICKET_PRICE);
        require(now < ticketingCloses);
        tickets.push(msg.sender);
                                    }
```

A LOTTERY ON THE BLOCKCHAIN

```
function drawWinner () public {
      require(now > ticketingCloses + 5 minutes);
      require(winner == address(0));
      bytes32 bhash=blockhash(block.number-1);
      bytes memory bytesArray = new bytes(32);
      for (uint i; i <32; i++){
          bytesArray[i] =bhash[i];
      bytes32 rand=keccak256(bytesArray);
      winner = tickets[uint(rand) % tickets.length];
      }
function withdraw () public {
      require(msg.sender == winner);
      msg.sender.transfer(address(this).balance);
  }
 fallback () payable public {
      buy();
  } }
```

SETTING DELAYED ACTIONS

```
constructor (uint duration) public {
        ticketingCloses = now + duration; }
```

- the constructor
  - inizializes the contract
- lottery closes after a time period
  - during this period people can buy tickets
- now is a reserved word
  - takes the current block timestamp
    - alias for block.timestamp
  - returns a UNIX timestamp
    - seconds after the UNIX epoch of *1970-01-01 00:00:00*
  - makes it easy to create delayed actions

LOTTERIES: DRAWING THE WINNER

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

- set a time delay between the ticket purchase period and the start of the winner drawing function
  - drawing will use blockhash as source of entrophy
  - if `drawWinner` and buy are invoked in parallel
    - the last buyer could know the blockhash, used by the drawing winner
    - avoid this delaying the drawing of the winner, after the end of the lottery

- ensure that the winner has not already been drawn, when invoked
  - use `address(0)`, the same as "0x0", an uninitialized address.

RANDOMENESS IN ETHEREUM

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

- Solidity is a deterministic language
  - all the validators (miners) must return the same result
  - cannot use built-in source of entropy to generate random numbers.
- a limited source of entropy
  - future blockhashes are unguessable for the user
  - take blockhash by the expression `blockhash(block.number – 1)`
  - apply the `kekkak256` function

RANDOMENESS IN ETHEREUM

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

- `block.number:` current block number
- `blockhash(uint blockNumber)`
  - hash of a given block number:
  - only works for the 256 most recent blocks
- globally available variables
  - special variables and functions which always exist in the global namespace
  - mainly used to provide information about the blockchain

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 send 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
- many alternate implementations exist today

GAMBLING: SATOSHI DICE

- implementation
  - submit a transaction with
    - a number from 0-65,535 (2^{16} = 65,536)
    - an ether amount
  - the game generates a random number in the same range by 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 = ~2x, 16,000 = ~4x
- in the real implementations, periodically publish the hash of the old seeds together the betting addresses to provide auditability

GAMBLING: SATOSHI DICE

```
pragma solidity ^0.6.0;
contract SatoshiDice {
    struct Bet {
        address user;
        uint block;
        uint cap;
        uint amount;
     }
   uint public constant FEE_NUMERATOR = 1;
   uint public constant FEE_DENOMINATOR = 100;
   uint public constant MAXIMUM_CAP = 100000;
   uint public constant MAXIMUM_BET_SIZE = 1e18;
   address payable owner;
   uint public counter;
   mapping(uint => Bet) public bets;
   event BetPlaced(uint id, address user, uint cap, uint amount);
   event Roll(uint id, uint rolled);
```

GAMBLING: SATOSHI DICE

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

function wager (uint cap) public payable {
    require(cap <= MAXIMUM_CAP);
    require(msg.value <= MAXIMUM_BET_SIZE);
    counter++;
    bets[counter] = Bet(msg.sender, block.number + 3, cap, msg.value);
    emit BetPlaced(counter, msg.sender, cap, msg.value);
}
```

GAMBLING: SATOSHI DICE

```
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);
    bytes32 bhash=blockhash(block.number-1);
    bytes memory bytesArray = new bytes(32);
    for (uint i; i <32; i++) {bytesArray[i] =bhash[i];}
    bytes32 random=keccak256(bytesArray);
    uint rolled = uint(random) % MAXIMUM_CAP;
    if (rolled < bet.cap) {
        uint payout = bet.amount * MAXIMUM_CAP / bet.cap;
        uint fee = payout * FEE_NUMERATOR / FEE_DENOMINATOR;
        payout -= fee;
        msg.sender.transfer(payout);
       }
    emit Roll(id, rolled);  delete bets[id]; }
```

GAMBLING: SATOSHI DICE

```
fallback () payable external {}

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

TYPE STRUCT

```
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

```
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
  - sintax similar to `struct`
- event logs:
  - registered on the blockchain
  - any JavaScript client can listen for events as callbacks
  - events can be indexed: search for specific events in the log


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

ACCEPTING WAGERS

```
function wager (uint cap) public payable {
    require(cap <= MAXIMUM_CAP);
    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
  - registeres as event on the blockchain, (field counter)
  - 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 three blocks after wagering
    - cannot guess the hashblock of 3 blocks in the future

ROLLING THE DICE

```
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);
    bytes32 random = keccak256(block.blockhash(bet.block), id);
```

- 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

ACCEPTING WAGERS

```
function wager (uint cap) public payable {
    require(cap <= MAXIMUM_CAP);
    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);}
```

- globally available variables
  - special variables and functions which always exist in the global namespace
  - mainly used to provide information about the blockchain

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

DATA LOCATION

```
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);
    bytes32 random = keccak256(block.blockhash(bet.block), id);
```

- bet should be stored on the blockchain
  - use the keyword storage to make the variable persistent

- Kekkak256
  - compute the Ethereum SHA-3 (Keccak-256) hash of the arguments

DATA LOCATION IN 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 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.

FUNDING AND KILLING

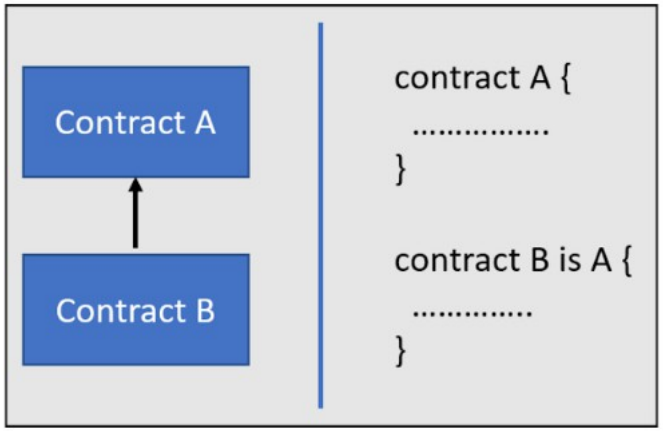
```
fallback () payable public {}

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
  - releases storage from the blockchain
  - remaining balance, sent it to the address passed as argument

CONTRACT INHERITANCE

- 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



CONTRACT INHERITANCE

```
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!!";} }
```

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

```
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);
          } }
```

FUNCTION MODIFIERS

```
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 code 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

CREATING CONTRACT: FACTORIES

```
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++; }
    }
```

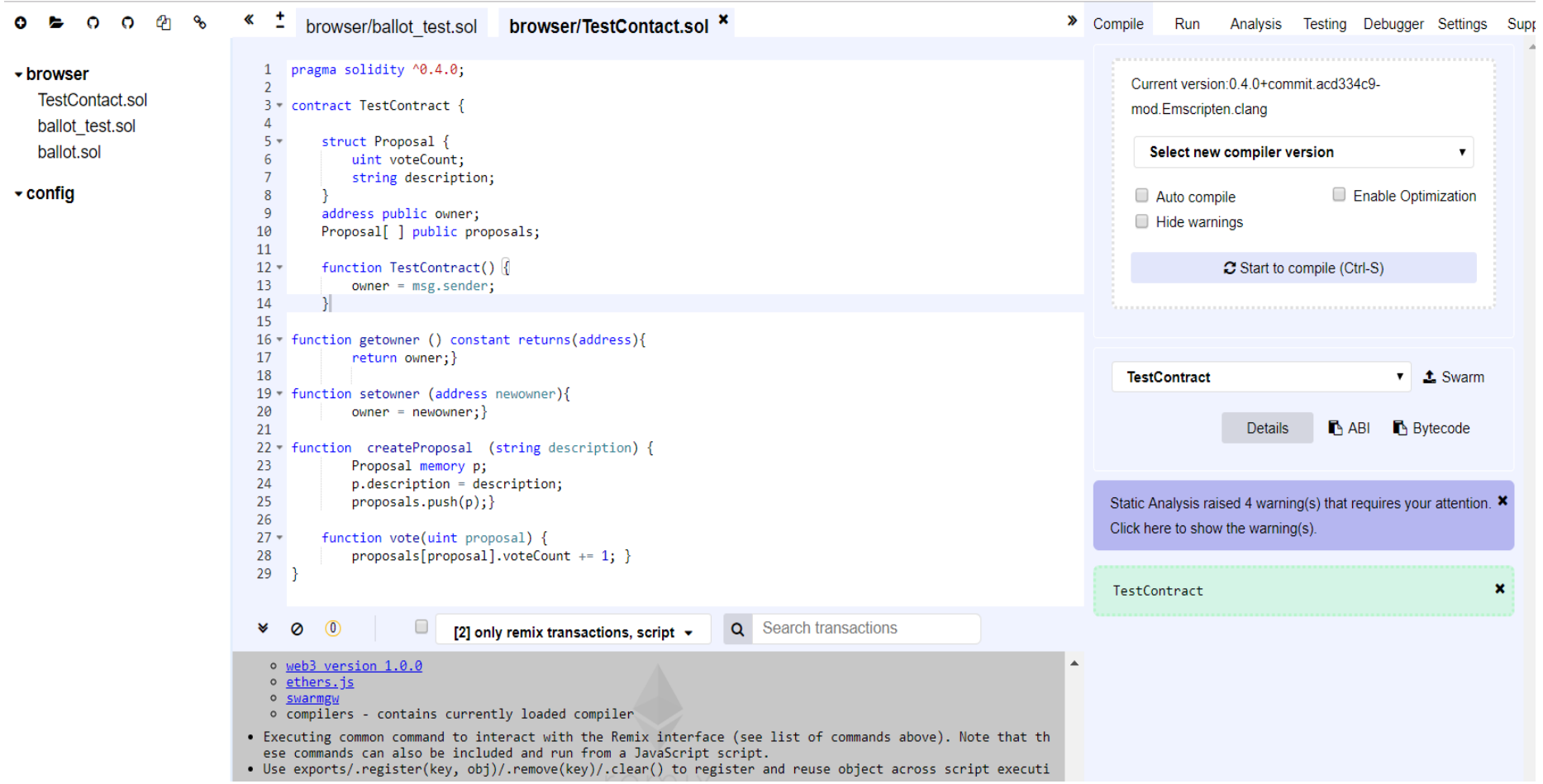
THE CONTRACT FACTORY

```
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));
    }}
```

HTTPS: //REMIX.ETHEREUM.ORG

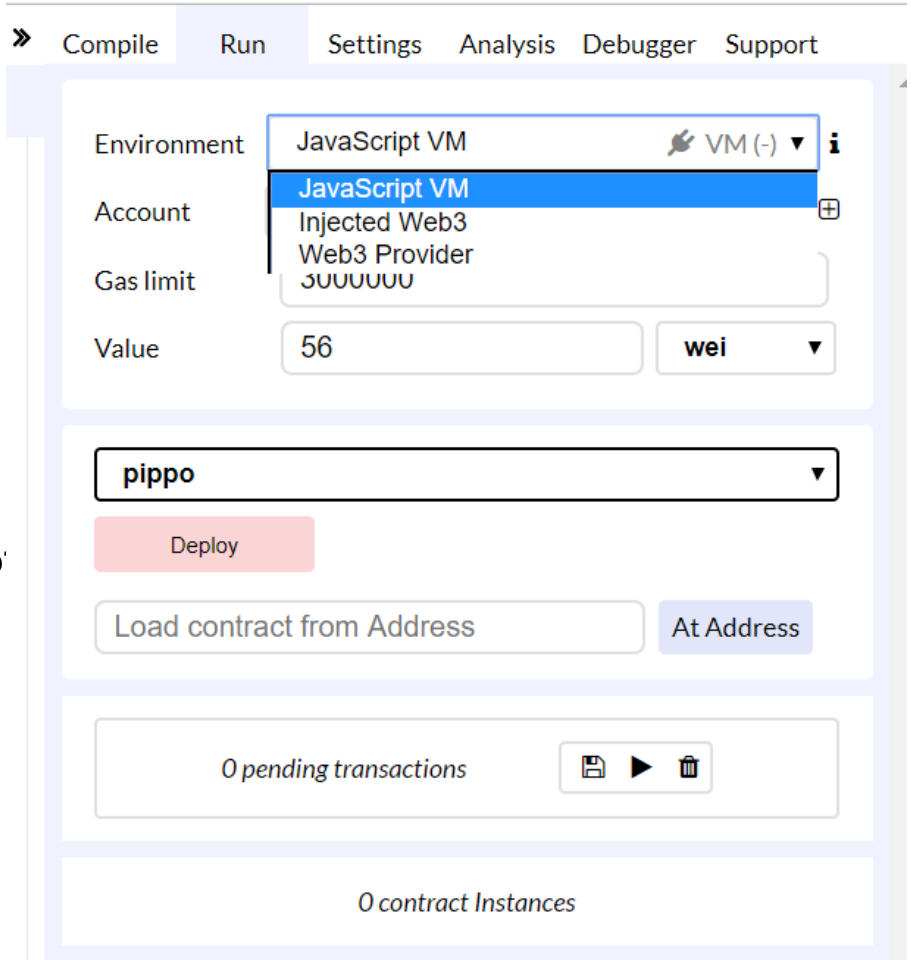
- Remix: a web based IDE for Solidity
  - simple to start playing with smart contracts
- structured in three different zones:
  - left file editor
  - central editor for creating a new contract in the browser
    - insert code here
  - right tools for the contact analysis and deployment
    - compile
    - deploy
    - debug
    - run
    - optimize gas consmption
    - ……

REMIX: A WEB BASED IDE FOR SOLIDITY



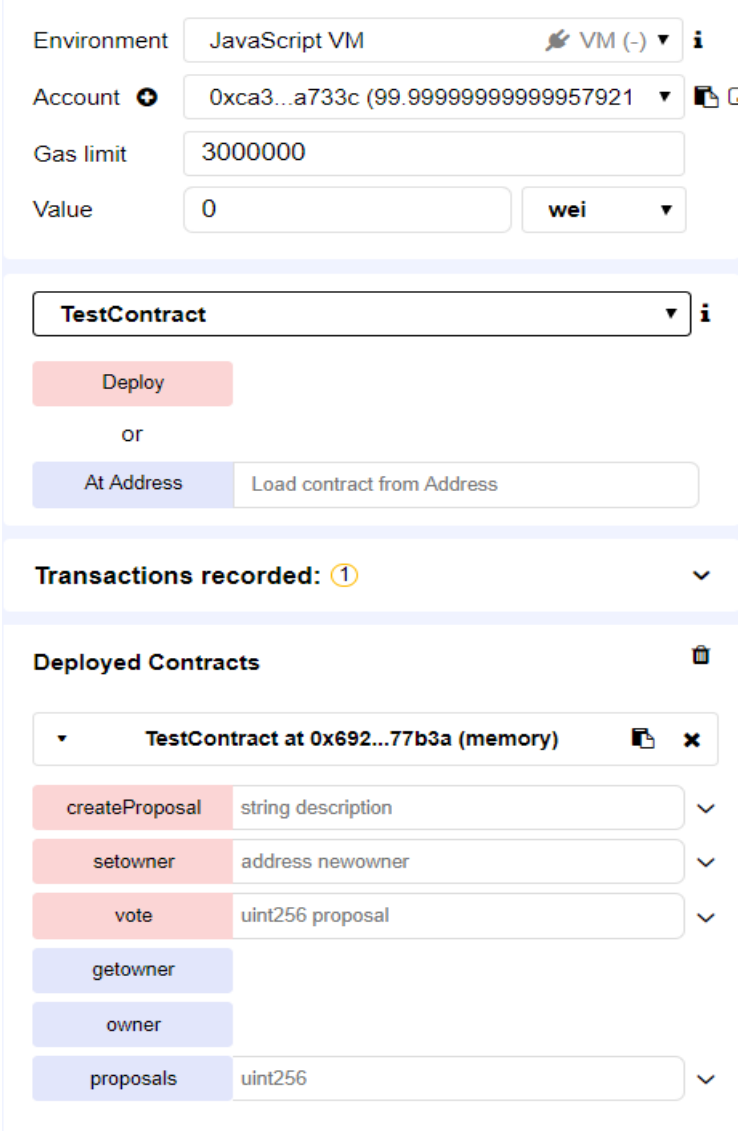
REMIX: A WEB BASED IDE FOR SOLIDITY

- run tab (on the right)
  - for the contract deployment
- 3 different deployment environments
  - JavaScript VM
    - a testnet run by the Remix IDE
  - injected web3
    - to deploy the contract on one o the Ethereum testnets
    - through MetaMask
  - web3 provider
    - to connect to a local testnet
- for the moment, choose JavaScript VM



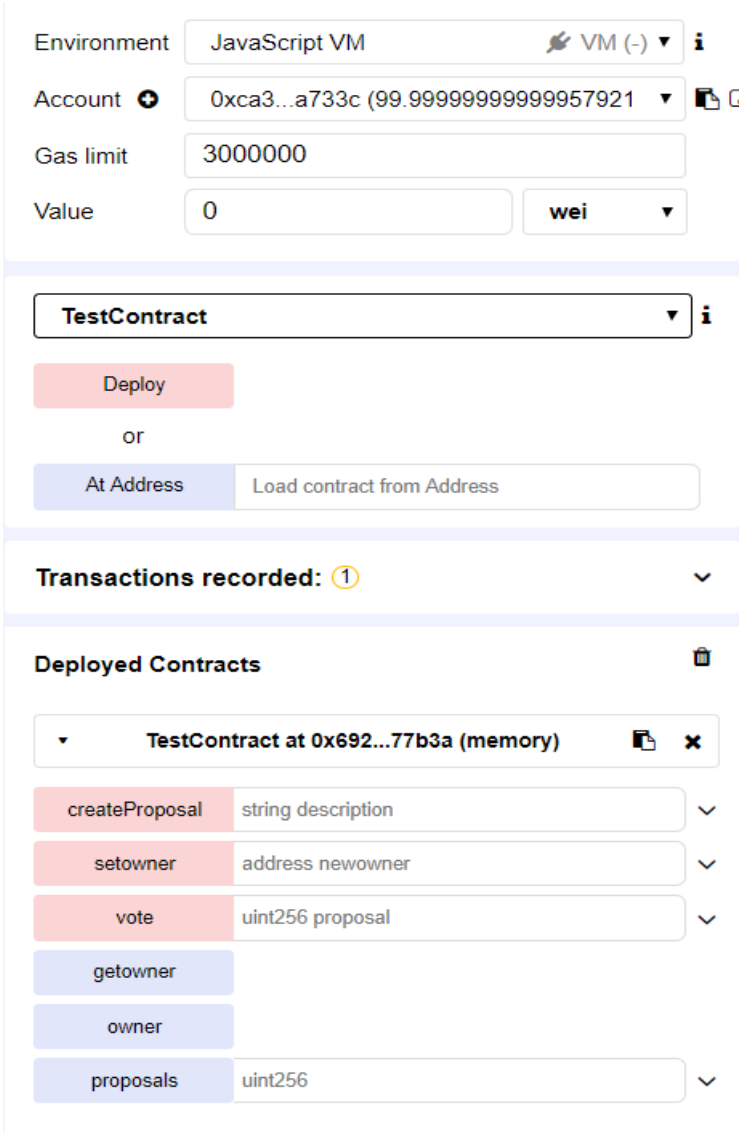
DEPLOYING A CONTRACT LOCALLY

- choose
  - an account from a predefined list
  - a gas limit
  - value
    - amount for this contract
  - the name of the contract to deploy
    - more contracts can be contained in the same file
- click on the deploy button
  - before making the deploy, be sure to have compiled the contract

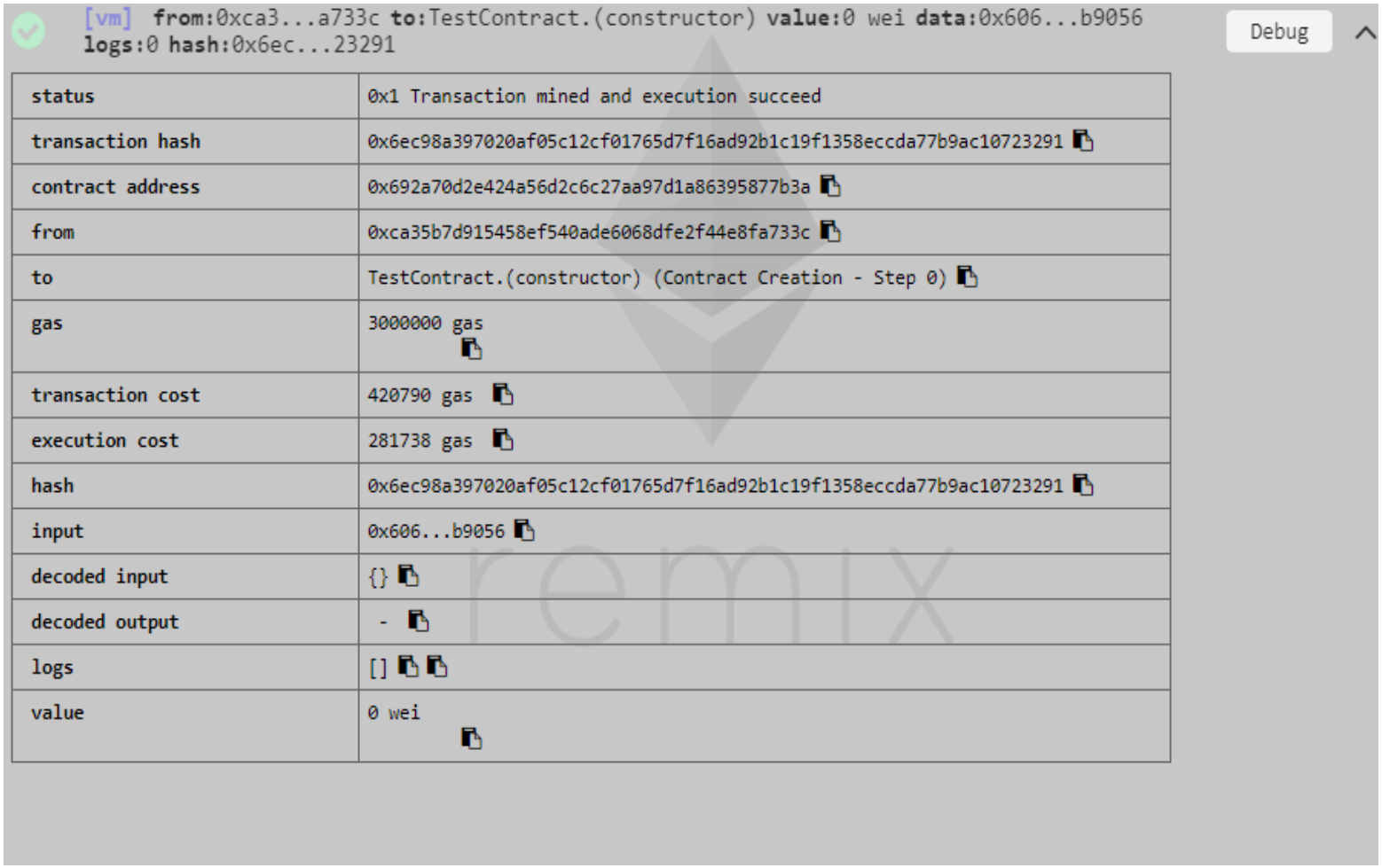


DEPLOYING A CONTRACT LOCALLY

- pink
  - functions which change the state
- pale blue
  - view/pure functions



REMIX: DEPLOYING A CONTRACT ON THE JS VM



If the contract has been inserted in a block from the miner it appears in a window below the edit window

MONITORING EVENTS

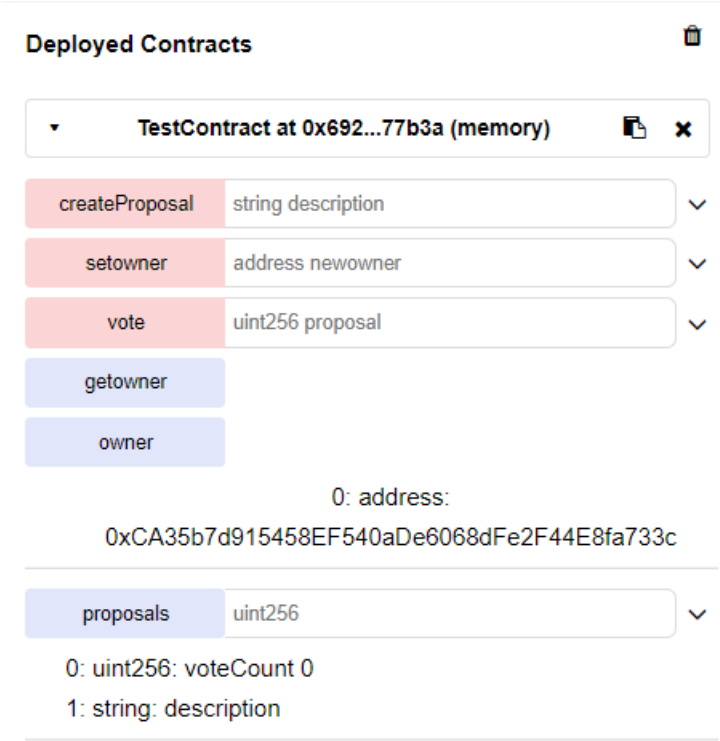
decoded output	<pre>{
	  "0": "uint256: 0"
	}</pre>

logs	<pre>[
	  {
	  "from": "0x692a70d2e424a56d2c6c27aa97d1a86395877b3a",
	  "topic": "0x9a6d8477b05c9f948f7027f8835f8afa2bcc4e2a2cc0eb931845c3114bbf9bfa",
	  "event": "IntegerAdded",
	  "args": {
	  "0": "0",
	  "1": "0",
	  "2": "0",
	  "x": "0",
	  "y": "0",
	  "result": "0",
	  "length": 3
	  }
	  }
	]</pre>

value	0 wei

Transaction Log: in the Logs field you can see the events logged by the transaction

AFTER THE DEPLOYMENT

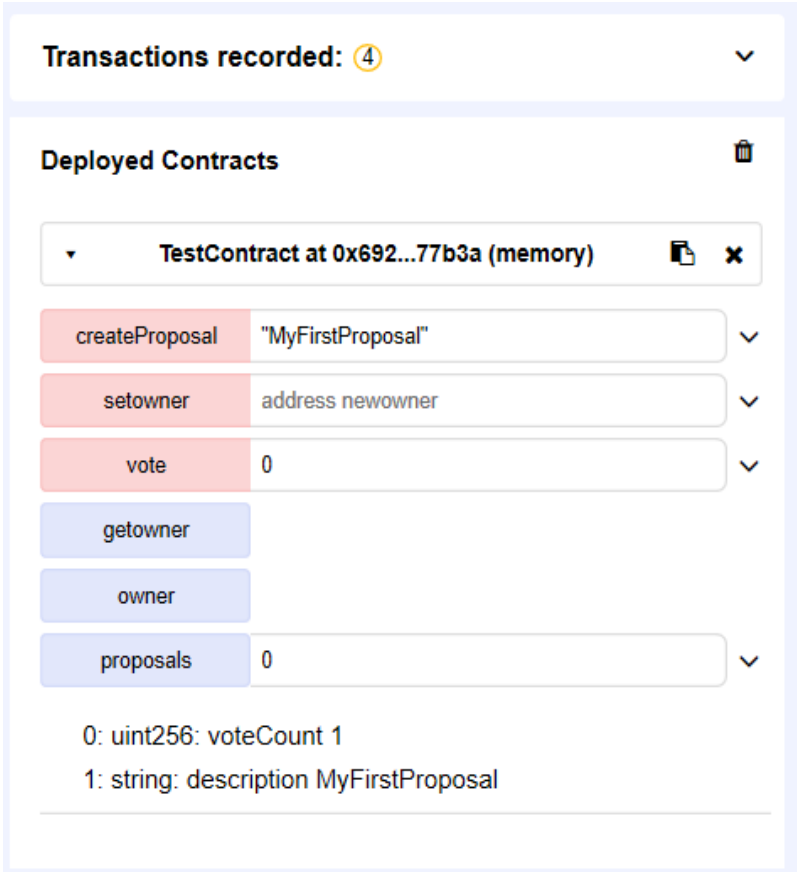


- contract deployed at the address: 0x143....
- pale buttons: query the current values of the public read-only fields of this contract.
  - the "owner" corresponds to the address that have created the contract
  - the array of proposals is empty

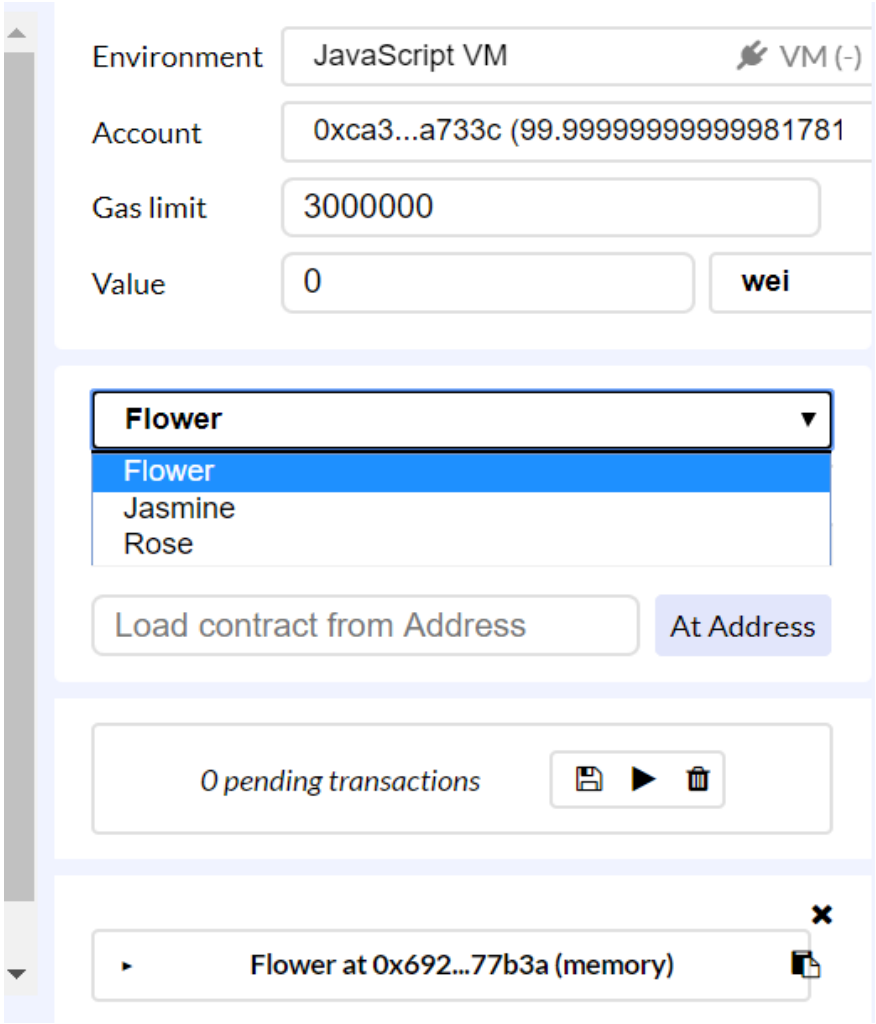
MAKING TRANSACTIONS

- invoke the constructor
  - MyFirstProposal
- invoke contrcat functions
- debug
  - see how the content of variables

    is changed pale blue button)



REMIX AND CONTRACT INHERITANCE



REMIX AND CONTRACT INHERITANCE