Lesson 18:
PROGRAMMING SMART CONTRACTS:
SOLIDITY AND REMIX

15/05/2019
ETHEREUM SMART CONTRACTS: RECAP

- A special account that stores executable code together with its associated data and an account balance on the blockchain.
- Identified by an address (a public key).
- Created by transactions.
- Transactions are used to interact with a contract on the blockchain:
  - By sending money to its account balance.
  - By executing code.
- To execute the code of a contract, a function call is sent to the contract:
  - Its parameters binary encoded in the data field of the transaction.
ETHEREUM SMART CONTRACTS: RECAP

- When a contract receives a message from another contract or a transaction from a user
  - can receive ether
  - can execute a function that is specified in the data field
- the contract can send money from its balance to other accounts or execute functions on other contracts.
- execution of the code takes place on all mining nodes in the network concurrently which reach consensus over the new state of the contract
REMIX: PLAYING WITH THE BLOCKCHAIN

• an in-browser Solidity editor: full IDE for:
  • edit contract code
  • test
  • deploy the code on different blockchains

• https://remix.ethereum.org

• functionalities: development of smart contracts
  • editor
  • debug a smart contract’s execution.
  • access the state and properties of already deployed smart contract.
  • debug already committed transactions.
  • analyze Solidity code to optimize it with respect to the graph consumption
  • Remix can be used to test and debug a contract on a real blockchain
    • with any tool which inject web3
pragma solidity ^0.4.0;

contract TestContract {

    struct Proposal {
        uint voteCount;
        string description;
    }

    address public owner;
    Proposal[ ] public proposals;

    function TestContract() {
        owner = msg.sender;
    }
}
function getowner () constant returns(address){
    return owner;}

function setowner (address newowner){
    owner = newowner;}

function createProposal (string description) {
    Proposal memory p;
    p.description = description;
    proposals.push(p);}

function vote(uint proposal) {
    proposals[proposal].voteCount += 1; }
}
structured in three different zones:

- **left** file editor (presents an already inserted contract ballot.sol)
- **central** editor for creating a new contract in the browser
  - insert code here
- **right** tools for the contract analysis and deployment
  - compile
  - deploy
  - debug
  - run
  - optimize gas consumption
  - …..
pragma solidity ^0.4.0;

contract TestContact {
    struct Proposal {
        uint voteCount;
        string description;
    }
    address public owner;
    Proposal[] public proposals;

    function TestContact() {
        owner = msg.sender;
    }

    function getowner () constant returns(address){
        return owner;
    }

    function setowner (address newowner){
        owner = newowner;
    }

    function createProposal (string description) {
        Proposal memory p;
        p.description = description;
        proposals.push(p);
    }

    function vote(uint proposal) {
        proposals[proposal].voteCount += 1;
    }
}
REMIX: A WEB BASED IDE FOR SOLIDITY

- **run tab** (on the right)
  - for the contract deployment
- different deployment environments
  - **JavaScript VM**
    - a testnet run by the Remix IDE itself
  - **injected web3**
    - to deploy the contract through MetaMask on one of the Ethereum testnets
  - **web3 provider**
    - to connect to a local testnet
- for the moment, choose **JavaScript VM**
REMIX: DEPLOYING A CONTRACT ON THE JS VM

- choose
  - an account for the contract from a predefined list
  - a gas limit
  - value
    - amount to be sent to 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
REMIX: DEPLOYING A CONTRACT ON THE JS VM

- pink
  - functions

- pale blue
  - variables
If the contract has been inserted in a block from the miner it appears in a window below the edit window.
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
AFTER THE DEPLOYMENT

• create a new proposal
  • MyFirstProposal

• vote for that proposal giving the index of the proposal

• see how the content of the proposals array is changed
  • click on the pale blue button “proposals” with input index 0
  • description should match what you typed in createProposals
  • the voteCount should be 0
SOLIDITY

- a contract oriented high-level language with a **static type system** that has a syntax very similar to javascript
  - a statically typed language, the data type of a variable must be specified.

- contract
  - collection of code (functions) + state residing in a block of the blockchain
  - state
  - similar to a class of an object oriented language

- It may contain
  - fields, functions
  - function modifiers,
  - data structures
  - events
  - ….
THE PRAGMA DIRECTIVE

- the pragma directive should be the first line of code in Solidity
  
  ```solidity
  pragma solidity ^0.4.0;
  ```

  specifies which Solidity compiler version to be used for the contract

- useful for languages in evolution, like Solidity
  - common for new compiler version to fail to compile older programs
  - the program should not be compiled with future compiler versions that might introduce changes incompatible with the code

- includes a major build (4), followed by a minor build number (0)
  - the caret (^) before the version number means that Solidity can use the latest build in the major version range
  - previous pragma matches version of the compiler $\geq 0.4.0$ and $< 0.5.0$
  - more complex expression can be specified to express range of versions
SOLIDITY TYPES

- **booleans**: boolean value (true, false)

- **unsigned (uint)**: its value must be non-negative.
  - declared in increments of 8 bits from int8 to uint256.
  - actually uint is an alias for uint256.

- **signed (int)**

- **address**: represents a 20 byte Ethereum address.
  
  0X0cE446255506E92DF41614C46F1d6df9Cc969183

- **fixed point**

- **byte array** (fixed and dynamic)

- **enum** user-defined type for enumerating discrete values
SOLIDITY STRUCTURED TYPES

- **struct**

```solidity
define struct Student {
    uint age;
    string name;
}
Student vitalik = Student (24, "Vitalik");
```

- **fixed-size byte arrays:** includes byte arrays from length 1 to 32.

```solidity
// array with a fixed length of 4 elements
uint [4] fixedArray;
// another fixed array, can contain 7 strings
string[7] stringArray;
```

- **dynamically-sized arrays:**

```solidity
// a dynamic Array - has no fixed size, can keep growing:
Student[ ] public students;
students.push(vitalik)
```
MAPPINGS

```solidity
pragma solidity ^0.4.12;

contract New {
    mapping (address => uint) public balances;
    uint public y;
    function set (address a, uint x) public {
        balances[a] = x;
    }
    function get (address a) public {
        y = balances[a];
    }
}
```

- like hash tables that provide one-move lookups and writes to a massive address space.
- when the map in the example is declared (before having actually written anything to it), it looks logically like:
  - a table with primary key of type address
  - all possible addresses exist
  - all the values corresponding to the keys (addresses) are initialized to 0.
- get returns always 0 if no previous call to set has been executed
when a contract is executed in the EVM, it has access to a small set of global objects.

- include the msg, block and tx objects.

object "msg" may represent
- the transaction call if the message is generated by a EOA
- the message call if the message is generated by a contract

provides some useful attributes
- the sender of the message
- the value sent with the message
- the remaining gas for a message call
SOLIDITY MSG

• provides some useful attributes
  • `msg.sender`:
    • address of the EOA or of the smart contract calling the current function.
    • function execution always needs to start with an external caller.
      • `msg.sender` is always defined
  • `msg.value`
  • `msg.gas`
  • `msg.data`
  • `msg.sig`
STORAGE VS MEMORY

- two places where you can store variables
  - storage
    - variables stored permanently on the blockchain. “like the hard disk”
    - stored in the state tree
    - expensive and should be used only when necessary
  - memory
    - temporary variables, erased between external function calls to your contract. “like the RAM”.
    - cleared after every transaction
    - cheap and should be used whenever possible.
- default:
  - variables declared outside of functions are by default storage
  - variables declared inside functions are memory and will disappear when the function call ends.
contract Airbud {
  // state variables forced to storage
  address[] users;
  mapping(address => uint) public balances;
  function yelp () public payable
  {
    // local variable defaults to storage
    address user = msg.sender;
  // local variable declared to memory
    uint8[3] memory ids = [1,2,3];
  }
}
FUNCTIONS: ACCESS MODIFIERS

```plaintext
function name (<parameter types>) <access modifier>
  [pure|view|payable]
  [returns (<return types>)]
```

- specify function visibility
  - **public** accessible from externally, from this contract and from inherited contracts
    - can be called both by message calls and internally by simple jumps
  - **external** like public functions, except they cannot be called from within the contract unless explicitly prefixed with the keyword this.
    - always called by message calls, even if internal
    - if f is external, f( ) will not work, this.f( ) works.
  - **private** accessible only from this contract and not by derived contracts
  - **internal** accessible only from this contract and contracts inheriting from it
FUNCTIONS: ACCESS MODIFIERS

function name (〈parameter types〉) 〈access modifier〉
[ 〈pure|view|payable〉
  〈returns (〈return types〉)〉]

• the terms internal and private are somewhat misleading
  • any function or data inside a contract is always visible on the public blockchain, meaning that 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.
• deploy following contract in Remix:

```solidity
pragma solidity^0.4.12;

contract Test {
    function test1(uint[20] a) public pure returns (uint){
        return a[10]*2;
    }

    function test2(uint[20] a) external pure returns (uint){
        return a[10]*2;
    }
}
```

<table>
<thead>
<tr>
<th>status</th>
<th>0x1 Transaction mined and execution succeed</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>0xca35b7d915458ef540ade6068dfe2f44e8fa733c</td>
</tr>
<tr>
<td>to</td>
<td>Test.test1(uint256[20]) 0x0dcd2f752394c41875e259e00bb44fd505297caf</td>
</tr>
<tr>
<td>gas</td>
<td>3000000 gas</td>
</tr>
<tr>
<td>transaction cost</td>
<td>25578 gas</td>
</tr>
<tr>
<td>execution cost</td>
<td>530 gas</td>
</tr>
</tbody>
</table>

Gas consumed: 530
• deploy following contract in Remix

```
pragma solidity^0.4.12;

contract Test {
    function test1(uint20 a) public pure returns (uint){
        return a[10]*2;
    }
}

function test2(uint20 a) external pure returns (uint){
    return a[10]*2;
}
```

<table>
<thead>
<tr>
<th>status</th>
<th>0x1 Transaction mined and execution succeed</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>0xca35b7d915458ef540ade6068dfe2f44e8fa733c</td>
</tr>
<tr>
<td>to</td>
<td>Test.test2(uint256[20]) 0x0dcd2f752394c41875e259e00bb44fd505297caf</td>
</tr>
<tr>
<td>gas</td>
<td>3000000 gas</td>
</tr>
<tr>
<td>transaction cost</td>
<td>25387 gas</td>
</tr>
<tr>
<td>execution cost</td>
<td>339 gas</td>
</tr>
</tbody>
</table>

Gas consumed: 339
• public functions consumes more gas than external functions. Why?

• in public functions, Solidity immediately copies array arguments to memory, and memory allocation is expensive
  • public functions are called by internal jumps in the code and the array arguments are passed internally by pointers to memory
  • arguments have to be located in memory

• external functions can read directly from calldata and reading from calldata is cheaper
  • the compiler doesn't need to allow internal calls, and so it allows arguments to be read directly from calldata, saving the copying step.
FUNCTIONS: STATE SPECIFIERS

- Affect the behaviour of a function:

  ```solidity
  function name (<parameter types>) <access modifier>
  [pure|view|payable]
  [returns (<return types>)]
  ```

- **view** functions: do not write to the storage

  ```solidity
  function f(uint a) view returns (uint) {
      return a * b; // where b is a storage variable
  }
  ```

- **pure** functions: neither read nor write to the storage

  ```solidity
  function add(uint256 a, uint256 b) internal pure returns (uint256) {
      uint256 c = a + b;
      assert(c >= a);
      return c;
  }
  ```
FUNCTIONS: STATE SPECIFIERS

- view and pure functions are used for data retrieval from the blockchain
  - methods that may be called immediately without a transaction being mined.
  - no charge for the gas used to execute the call.

- actions modifying the blockchain
  - contract creation
  - change of the state of the contract
  - such actions require to be submitted as a new transaction
    - transactions are picked up by miners, functions are executed and state change incorporated into new blocks in the blockchain.
VIEW FUNCTIONS

```solidity
mapping (address => balance) balances;

function setMyBalance(uint _myBalance) public {
    balances[msg.sender] = _myBalance;
    // ^ The syntax for storing data in a mapping is just like with arrays
}

function whatIsMyBalance() public view returns (uint) {
    // Retrieve the balance stored in the sender's address
    // Will be `0` if the sender hasn't called `setMyBalance` yet
    return balances[msg.sender];
}
```

- use msg.sender to guarantee security
  - transaction may be initiated only if the user owns the private key
  - the only way someone can modify someone else's balance would be to steal the private key associated with their Ethereum address.
PAYABLE FUNCTIONS

• a special type of function that can receive Ether.
  • require a certain payment to the contract in order to execute a function.

• possible because in Ethereum
  • the money (Ether)
  • the data (transaction payload),
  • the contract code itself

all live in the same system

• mark the function as payable
  • if it is not marked payable and you try to send Ether to it, the function will reject your transaction.
PAYABLE FUNCTIONS

```solidity
pragma solidity ^0.4.8;

contract OnlineStore {
    function buySomething() external payable {
        // Check to make sure 0.001 ether was sent to the function call:
        require(msg.value == 0.001 ether);
        // If so, some logic to transfer the digital item to the caller of the
        // Function:
        transferThing(msg.sender);
    }
}
```

- the transaction is the envelope
  - the parameters you send to the function call are the contents of the letter
  - adding a value is like putting cash inside the envelope
    - the `msg.value` field reports this cash
- after you send Ether to a contract, it gets stored in the contract's Ethereum account,
EXCEPTION HANDLING AND ATOMICITY

• a contract call can terminate and return an error

• assert, require: to handle exceptional situations
  • operate in the same way, evaluating a condition and stopping execution with an error if the condition is false

• when a contract terminates with an error
  • all the state changes (changes to variables, balances, etc.) are reverted
  • this happens for all the chain of contract calls if more than one contract was called.

• this ensures the atomicity of transactions
  • they either complete successfully
  • or have no effect on state and are reverted entirely.
EXCEPTION HANDLING: ASSERT

- used when the outcome is expected to be true
  - used to test internal conditions
  - check overflow/underflow
  - divide by 0...
- make sure that the contract will not fall into an invalid state
  - throws an error and stops execution if some condition is not true
- revert all the changes made
- consumes all the gas of the transaction

```solidity
pragma solidity ^0.4.16;

contract Test {
    function add(uint256 a, uint256 b) internal pure returns (uint256 c)
    {
        c = a + b;
        assert(c >= a);
        return c;
    }
}```
EXCEPTION HANDLING: REQUIRE

- used when testing inputs
  - function arguments or transaction fields
    - `require(input < 20)`
  - setting expectations for those inputs
- throws an error and stops execution if some condition is not true
- revert all the changes made
- consume the gas used up to the point of failure.
  - the remaining gas will be refunded

```solidity
pragma solidity ^0.4.16;

contract Auction{
  function bid() payable external {
    address highestBidder;
    uint highestBid;
    require(msg.value >= highestBid);
    ...
pragma solidity ^0.4.16;

contract Test {
    function run(uint8 i) public pure {
        uint8 total = 0;

        for (uint8 j = 0; j < 10; j++)
            total += j;

        assert (i < 20);
        require (i < 10);

        for (j = 0; j < 10; j++)
            total += j;
    }
}
Testing Assert and Require in Remix

Environment: JavaScript VM
Account: 0xca3...a733c (99.99999999999987631)
Gas limit: 3000000
Value: 0 wei

Test
Deploy
Load contract from Address
At Address

0 pending transactions

Test at 0x692...77b3a (memory)
run 8
### Testing Assert and Require in Remix

<table>
<thead>
<tr>
<th>from</th>
<th>0xca35b7d915458ef540ade6068dfe2f44e8fa733c</th>
</tr>
</thead>
<tbody>
<tr>
<td>to</td>
<td>Test.run(uint8) 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a</td>
</tr>
<tr>
<td></td>
<td>23351 gas (Cost only applies when called by a contract)</td>
</tr>
<tr>
<td></td>
<td>1887 gas (Cost only applies when called by a contract)</td>
</tr>
<tr>
<td>input</td>
<td>0xc4e5557a000000000000000000000000000000000000000000000000000008</td>
</tr>
</tbody>
</table>
| decoded input | ```
``` |"uint8 i": 8

Test #1 - run(8): Function runs successfully - 1887 gas consumed.
### Testing Assert and Require in Remix

<table>
<thead>
<tr>
<th>from</th>
<th>0xca35b7d915458ef540ade6068dfe2f44e8fa733c</th>
</tr>
</thead>
<tbody>
<tr>
<td>to</td>
<td>Test.run(uint8) 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a</td>
</tr>
<tr>
<td>transaction cost</td>
<td>22540 gas (Cost only applies when called by a contract)</td>
</tr>
<tr>
<td>execution cost</td>
<td>1076 gas (Cost only applies when called by a contract)</td>
</tr>
<tr>
<td>input</td>
<td>0xc4e5557a0000000000000000000000000000000000000000000000f</td>
</tr>
</tbody>
</table>
| decoded input   | `{  
|                |   "uint8 i": 15  
|                | }` |

Test #2 - run(15): Function passes assert, fails at require. Only first loop is run - 1076 gas consumed.
TESTING ASSERT AND REQUIRE IN REMIX

<table>
<thead>
<tr>
<th>from</th>
<th>0xca35b7d915458ef540ade6068dfe2f44e8fa733c</th>
</tr>
</thead>
<tbody>
<tr>
<td>to</td>
<td>Test.run(uint8) 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a</td>
</tr>
<tr>
<td>transaction cost</td>
<td>3000000 gas (Cost only applies when called by a contract)</td>
</tr>
<tr>
<td>execution cost</td>
<td>2978536 gas (Cost only applies when called by a contract)</td>
</tr>
<tr>
<td>input</td>
<td>0xc4e5557a0000000000000000000000000000000000000000000000000000000000000019</td>
</tr>
<tr>
<td>decoded input</td>
<td>{ &quot;uint8 i&quot;: 25 }</td>
</tr>
</tbody>
</table>

Test #3 - run(25): Function fails at assert. Again, only first loop is run - 2978536 gas consumed
FUNCTION MODIFIERS

• like a function decorator
  • creates additional features on function
  • apply some restriction on function

• to define a modifier
  • replace the keyword function with modifier referring the name of the function
  • in the modifier body: insert underscore _ into modifier definition, to represent a piece of code
What is the problem with this contract?

- the transfer function doesn’t ensure that the sender has enough in their account.

- of course we can fix this by introducing a conditional into the body of the function.
FUNCTION MODIFIERS

```solidity
modifier only_with_at_least(uint x) {
    require (balances[msg.sender] >= x); _;
}

function transfer(uint _amount, address _dest) public only_with_at_least(_amount) {
    balances[msg.sender] -= _amount;
    balances[_dest] += _amount;
}
```

- 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
FUNCTION MODIFIERS

• modifier reuse:
  • reuse for function vote, allowing only anyone with a balance more than 1000 to vote on some issue.

• reuse the same modifier for different functions
pragma solidity^0.4.12;

contract Votazioni

{   mapping (address => uint) public balances;
    mapping (address => uint) public votes;
    address public owner;
    constructor () public {
        owner=msg.sender;
        balances[owner] = 1000000;
    }

    modifier only_with_at_least(uint x) {
        require (balances[msg.sender] >= x); _;
    }

    modifier only_with_under(uint x) {
        require (balances[msg.sender] < x); _;
    }

    modifier only_when_voted {
        require (votes[msg.sender] != 0); _;
    }
function clear_un deserved_vote () public
only_with_under(1000) only_when_voted {
    delete votes[msg.sender];
}

function transfer(uint _amount, address _dest) public
only_with_at_least(_amount) {
    balances[msg.sender] -= _amount;
    balances[_dest] += _amount;
    clear_un deserved_vote();
}

function vote(uint _opinion) public
only_with_at_least(1000) {
    votes[msg.sender] = _opinion;
}
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.
- Multiple inheritance.
pragma solidity ^0.4.8;

contract Flower {
    address owner;
    string flowerType;

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

    function water() constant returns (string){
        return "ohhh, thanks I love water!";
    }
}

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

contract Jasmine is Flower ("Jasmine") {
    function smell() constant returns (string) {
        return "Mmmmm, smells godd!!";
    }
}
CONTRACT INHERITANCE

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)
CONTRACT INHERITANCE

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

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

- Rose at 0x0dc...97caf (memory)
  - pick
  - water
**CONTRACT INTERFACES**

- for your contract to talk to another contract on the blockchain that you don't own
  - import the contract OR
  - use an abstract contract/interface
- use a contract interface
  - like a contract skeleton
  - only declare the functions that you want to interact with not other functions or state variables
  - do not define the function bodies, simply end the function declaration with a semi-colon (;)

```solidity
pragma solidity ^0.4.4;

contract A{
    function f1(bool arg1, uint arg2) public returns (uint);
}
```
To use the functions in the interface, you have

- to know the address of the called contract on Ethereum
- to instantiate a local proxy to that contract
- then you can call the function through the proxy

```solidity
pragma solidity ^0.4.4;

contract A{
    function f1(bool arg1, uint arg2) public returns (uint);
}

contract YourContract{
    function doYourThing(address AddressOfA) public returns (uint)
    {
        A myA = A (AddressOfA);
        return myA.f1(true, 3);
    }
}
```
every contract can have a **fallback function**:

- one unnamed function: at most one for each contract
- cannot have arguments, nor return anything
- acts as a default function to execute
  - when no other functions can match the transaction call
  - when a transaction is sent to a contract sending only Ether (no data)
- convenient for users: they can simply send ether to an address and the function will execute.

```
function () payable {
    buy();
}
```

sending ether to the contract will execute the buy function
EVENTS

a way to communicate that something happened on the blockchain

• the app in the front-end, can be 'listening' for certain events and take action when they happen.

• the event is registered in a log that can be queried

```
pragma solidity ^0.4.0;

contract Event {
    // declare the event
    event IntegersAdded(uint x, uint y, uint result);

    function add(uint _x, uint _y) public returns (uint) {
        uint result = _x + _y;
        // fire an event to let the app know the function was called:
        emit IntegersAdded(_x, _y, result);
        return result;
    }
}
```
In Remix you can see the logged events in the transaction log.
**MONITORING EVENTS**

Transaction Log: information about the transaction

<table>
<thead>
<tr>
<th>status</th>
<th>0x1 Transaction mined and execution succeed</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
<td>0xca35b7d915458ef540ade6068dfe2f44e8fa733c</td>
</tr>
<tr>
<td>to</td>
<td>Event.add(uint256,uint256) 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a</td>
</tr>
<tr>
<td>gas</td>
<td>3000000 gas</td>
</tr>
<tr>
<td>transaction cost</td>
<td>23450 gas</td>
</tr>
<tr>
<td>execution cost</td>
<td>1922 gas</td>
</tr>
<tr>
<td>input</td>
<td>0x771602f70000000000000000000000000000000000000000000000000000000000000000000000000000000000</td>
</tr>
<tr>
<td>decoded input</td>
<td>`{</td>
</tr>
<tr>
<td></td>
<td>&quot;uint256 _x&quot;: &quot;0&quot;,</td>
</tr>
<tr>
<td></td>
<td>&quot;uint256 _y&quot;: &quot;0&quot;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
MONITORING EVENTS

Transaction Log: in the Logs field you can see the events logged by the transaction
• The balance of an address can be queried using the property `balance`

• Ether (in units of wei) can be sent to an address by three ways
  
  `address.transfer(value)`
  
  `address.send(value)`
  
  `address.call.value(value)()`

```
pragma solidity ^0.4.8;
contract RevenueSharing {
    address x = 0x123;
    address myAddress = this;

    function transferMoney() public{
        if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
    }
}
```
tradeoffs between `send()`, `transfer()`, `call.value()`

- `send` and `transfer` are considered safe against reentrancy.
  - these methods trigger code execution, but the called contract is only given an amount of 2,300 gas which is currently only enough to log an event.
  - `transfer(y)` equivalent to `require(x.send(y))`
    - it will automatically revert if the send fails.

- `call.value(y)()`
  - send the provided Ether and trigger code execution.
  - the executed code is given all available gas for execution making this type of value transfer unsafe against reentrancy.
  - reason for the DAO attack