P2P Systems and Blockchains
Spring 2018,
instructor: Laura Ricci
laura.ricci@unipi.it
Lesson 20:
PROGRAMMING SMART CONTRACTS:
SOLIDITY
23/05/2018
ETHEREUM SMART CONTRACTS

- 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

- Every time 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 using a proof-of-work algorithm.
a contract oriented high-level language with a static type system that has a syntax very similar to javascript

main construct in Solidity is the contract, which is similar to a class, can contain
- fields, functions
- function modifiers,
- struct types
- enum types
- ...

- a statically typed language, the data type of a variable must be specified.
**THE PRAGMA DIRECTIVE**

- "version pragma": first line of all Solidity contracts
  
  \[
  \text{pragma solidity } ^0.4.0;
  \]

  specifies which Solidity compiler version to be used for the contract

- the program should not be compiled with future compiler versions that might introduce changes incompatible with the code

- 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 VARIABLES AND TYPES

- **State variables**
  - permanently stored in contract storage.
  - they are written to the Ethereum blockchain and remain there forever.

- **Solidity types**
  - **Booleans**: stores a boolean value (true, false)
  - **Integers**: different subtypes, with different lengths
    - unsigned (uint): its value must be non-negative. actually an alias for uint256
    - signed (int)
  - **Address**: represents a 20 byte Ethereum address.
    - 0x0cE446255506E92DF41614C46F1d6df9Cc969183
SOLIDITY STRUCTURED TYPES

- **struct**

  ```solidity
  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

```
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
STORAGE VS MEMORY

- two places where you can store variables
  - storage
  - memory.
- Storage
  - variables stored permanently on the blockchain. “like the hard disk”
- Memory
  - Temporary variables, erased between external function calls to your contract. “like the RAM”.
- Default:
  - variables declared outside of functions are by default storage
  - variables declared inside functions are memory and will disappear when the function call ends.
- Why declaring explicitly the storing option?
STORAGE VS MEMORY

```solidity
pragma solidity ^0.4.12;

Sandwich[ ] sandwiches;

function eatSandwich(uint _index) public {
    Sandwich mySandwich = sandwiches[_index];...
    mySandwich.status = "Eaten!";
}
```

- the Solidity compiler returns a warning telling that you should explicitly declare “storage” or “memory” here.

```solidity
Sandwich storage mySandwich = sandwiches[_index];
mySandwich.status = "Eaten!";
```

- `mySandwich` is a pointer to `sandwiches[_index]` in storage, and the status of `sandwiches[_index]` is permanently modified on the blockchain.

```solidity
Sandwich memory anotherSandwich = sandwiches[_index + 1];
autherSandwich.status = "Eaten!";
```

- “anotherSandwich” is a copy of the data in volatile memory and will not change the status of the blockchain.
SOLIDITY TYPES

• some global variables and functions which are accessible by all functions
  • information about the state of the blockchain/transactions

• "msg" object that provides information about the message sent to a contract in a function call
  • the sender of the message
  • the remaining gas for a message call
  • the value sent with the message

• msg.sender:
  • the address of the EOA (or smart contract) who has called the current function.
  • a contract will just sit on the blockchain doing nothing until someone calls one of its functions,
    • msg.sender is always defined
  • function execution always needs to start with an external caller.
FUNCTIONS: ACCESS MODIFIERS

• General definition of a Solidity function:

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

• Access Modifier:
  • **public**: accessible from this contract, inherited contracts and externally,
    • can be called both by message calls and internally by simple jumps
  • **external**: cannot be accessed internally, only externally.
    • always called by message calls, even if internal
    • if f is external, f() will not work, this.f() works and generates a message.
  • **private**: accessible only from this contract
  • **internal**: accessible only from this contract and contracts inheriting from it

• Why both public and external?
  • *from the docs: external functions are sometimes more efficient when they receive large arrays of data.*
PUBLIC AND EXTERNAL

• 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

```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.test2(uint256[20]) 0x0dcd2f752394c41875e259e00bb44fd505297caf</td>
</tr>
<tr>
<td>gas</td>
<td>30000000 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 AND EXTERNAL

• Public functions consume 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.
EXCEPTION HANDLING: ASSERT

Assert (condition)

- to check if something completely unexpected happens and make sure that the contract will not fall into an invalid state
  - check overflow/underflow
  - divide by 0...
- 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**

**Require (condition)**

- a guard against transactions that you would not want to execute although the function would allow it if `require()` was omitted
  - validate user inputs, `require(input<20);`
- throws an error and stop 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

Test

Deploy

Load contract from Address
At Address

0 pending transactions

Test at 0x692...77b3a (memory)
from | 0xca35b7d915458ef540ade6068dfe2f44e8fa733c
---|---
to | Test.run(uint8) 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
transaction cost | 23351 gas (Cost only applies when called by a contract)
execution cost | 1887 gas (Cost only applies when called by a contract)
input | 0xc4e5557a0000000000000000000000000000000000000000000000000000000000000008
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>0xc4e5557a00000000000000000000000000000000000000000000000f</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.
Test #3 - run(25): Function fails at assert. Again, only first loop is run - 2978536 gas consumed
FUNCTIONS: STATE SPECIFIERS

- General definition of a function:

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

- View and pure attributes replace (from version 0.4.16) the constant attribute, deprecated in the last releases
  - 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

- 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 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.
**VIEW FUNCTIONS**

- use `msg.sender` to guarantee security

```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];
}
```

- 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 value field:**
  - 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
- after you send Ether to a contract, it gets stored in the contract's Ethereum account,
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
modifier only_with_at_least(uint x) {
    require (balances[msg.sender] >= x); _;
}

function vote(uint _opinion) public only_with_at_least(1000) {
    votes[msg.sender] = _opinion;
}

• 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 _opin ion) public
only_with_at_least(1000) {
    votes[msg.sender] = _opin ion;
}
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, if internal

Multiple inheritance

```solidity
contract A {
    .......... 
}

contract B is A {
    .......... 
}
```
pragma solidity ^0.4.8;

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

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

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

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

<table>
<thead>
<tr>
<th>Environment</th>
<th>JavaScript VM</th>
<th>VM (-)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>0xca3...a733c (99.99999999999981781)</td>
<td></td>
</tr>
<tr>
<td>Gas limit</td>
<td>3000000</td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td>0</td>
<td>wei</td>
</tr>
</tbody>
</table>

#### Flower
- 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);
}
```
CONTRACT INTERFACES

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);
    }
}
```
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.

```solidity
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;
    }
}
```
MONITORING EVENTS

In Remix you can see the logged events in the transaction log.
Transaction Log: information about the transaction
### MONITORING EVENTS

**decoded output**

```json
{
    "0": "uint256: 0"
}
```

<table>
<thead>
<tr>
<th>logs</th>
</tr>
</thead>
</table>
| [<br>```json
[
    {
        "from": "0x692a70d2e424a56d2c6c27a97d1a86395877b3a",
        "topic": "0x9a6d8477b05c9f948f7027f8835f8afa2bcc4e2a2cc0eb931845c3114bbf9bfa",
        "event": "IntegerAdded",
        "args": {
            "0": "0",
            "1": "0",
            "2": "0",
            "x": "0",
            "y": "0",
            "result": "0",
            "length": 3
        }
    }
]<br>`
```
| value | 0 wei |

Transaction Log: in the Logs field you can see the events logged by the transaction
TRANSFERRING ETHER BETWEEN ADDRESSES

- The balance of an address can be queried using the property `balance`.

- Ether (in units of wei) can be sent to an address using the `transfer` function.

- If `x` is a contract address, its code, i.e. its `fallback function` is executed together with the transfer call.

- If the execution runs out of gas or fails in any way, the Ether transfer will be reverted and the current contract will stop with an exception.

- The `send` call instead returns instead a boolean value.

```solidity
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);
  }
}
```
FALLBACK FUNCTIONS

- A contract may have precisely one unnamed function, which cannot have arguments, nor return anything. This function is named “fallback function”.

- Fallback functions are executed if a contract is called and no other function matches the specified function identifier, or if no data is supplied.

- These functions are also executed whenever a contract would receive plain Ether, without any data.
THE REVENUE SHARING CONTRACT

- some of the concepts we have shown in the previous slide are presented with respect to a real contract

- The Revenue Sharing Contract
  - a list of addresses is given when the contract is deployed.
  - anyone can send a certain amount of money to the contract
  - this amount of money is equally distributed to the addresses on the list.
pragma solidity ^0.4.8;

contract RevenueSharing {
    address public creator;
    mapping(uint => address) public shareholders;
    uint public numShareholders;

    event Disburse(uint _amount, uint _numShareholders);

    constructor (address[] addresses) public {
        creator = msg.sender;
        numShareholders = addresses.length;
        for (uint i=0; i<addresses.length; i++) {
            shareholders[i] = addresses[i];
        }
    }

    function shareRevenue() payable public returns (bool success) {
        uint amount = msg.value / numShareholders;
        for (uint i=0; i<numShareholders; i++) {
            if (!shareholders[i].send(amount)) revert();
        }
        emit Disburse(msg.value, numShareholders);
        return true;
    }

    function kill() public {
        if (msg.sender == creator) selfdestruct(creator);
    }
}