ethereum vienna

Workshop
From Idea to Contract

Workshops

Workshop #1: Contract Development for Beginners

Requirements: Basic Understanding of Ethereum

Solidity Basics

Workshop #2: From Idea to Contract

Requirements: Basic Understanding of Solidity

Mapping the real world to ethereum concepts

Advanced Solidity

Workshop #3: From Contract to DApp

Requirements: Basic Understanding of Solidity, HTML/JS, node.js

Interfacing with Ethereum using web3.js

Auxiliary Technologies: IPFS, Whisper and Swarm

Agenda

1. Mapping real world concepts to solidity
2. “Advanced” Solidity
3. Standard Contracts
4. **Developing the Contract**
5. Testing
6. **Testing the Contract**
7. Deployment

Groups

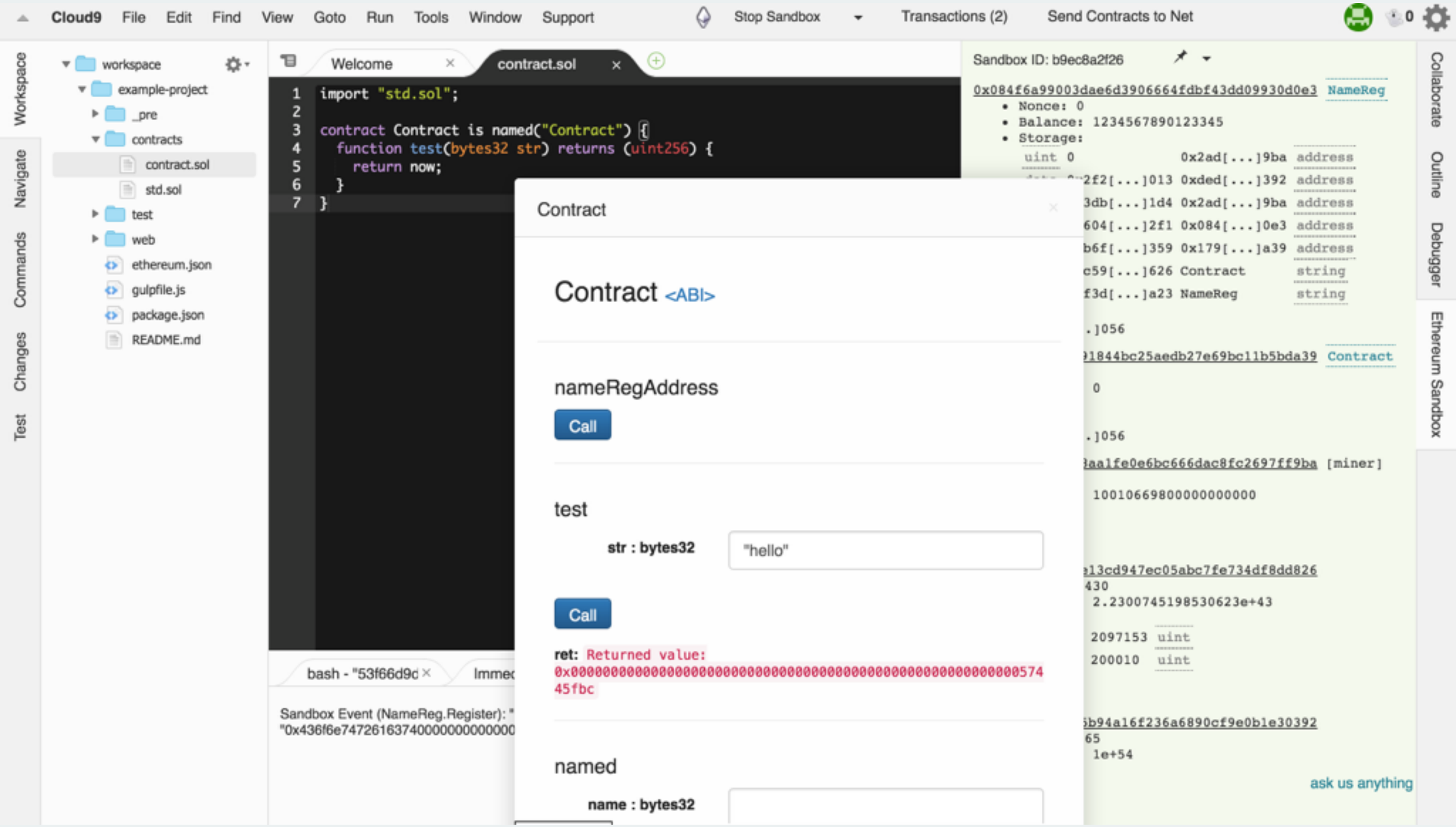
3 People per group (max 7 groups)

Working on the same contract

Requirements:

- Knowledge of basic solidity (WS1)
- No IDE required - ethereum studio might be useful
- geth, solc, node.js

Ethereum Studio



Real World

The thing that actually matters

Agreements can be easily broken

RW Actors

Contracts

Only a image of the state of the real world

Change in RW not reflected in contract

Change in contract not reflecting change in RW

Only on-blockchain data is fully secured

ether, tokens, owners of blockchain assets

Real World -> Contracts

RW Actors represented by (external) accounts. (Not sybil safe)

RW Information cannot be trusted in a trustless way

Many RW Applications can never be fully trustless

=> minimise required trust (multisig, incentives, etc.)

Real World -> Contracts

Mapping

When something has a non-incrementing identifier

All Fields 0 is either

the default state for some key

or an invalid state

cannot be enumerated

Ideal when you need to map from addresses to something

Real World -> Contracts

Array

When the index can be derived from the id

Incrementing id

When you need to enumerate a collection

Sometimes an array and a mapping to the same data can be useful

Real World -> Contracts

- Events
  - Everything something in the RW needs to react to
  - Only way to be light client friendly

Real World -> Contracts

external data

Trusted Data Feed

Multisig

3rd party with minimised trust (like oraclize for web resources)

Solidity

Solidity Contract Interaction

Interface

```
contract token {
    mapping (address => uint) public coinBalanceOf;
    function token() {}
    function sendCoin(address receiver, uint amount) returns(bool sufficient) {  }
}
```

Usage

```
function test() returns (uint) {
    address v = 0x13389247f732748971389738714738974982347 9;
    /* cast address to contract type */
    token t = token(v);
    /* call sendCoin function */
    t.sendCoin(0xffffffffffffffffffffffffffffffffffffffff, 1);
    /* call sendCoin function with 10000 gas limit and send one ether */
    t.sendCoin.gas(10000).value(1 ether)(0xffffffffffffffffffffffffffffffffffffffff, 1);
    /* calling autogenerated getter */
    return t.coinBalanceOf(0xffffffffffffffffffffffffffffffffffffffff);
}
```

Call Types

send	sends a message with 0 gas
call	sends a message with gas
callcode	call code in the current context
delegatecall	like callcode, but preserves sender and value
create	creates contract and runs init code

Solidity Contract Interaction

Creation

```
contract a { }

contract b {
    a v;

    function b () {
        v = new a();
    }
}
```

Solidity Import

```
import "filename";
import * as symbolName from "filename";
import {symbol1 as alias, symbol2} from "filename";
import "filename" as symbolName;
```

Solidity Libraries

Used to share code between contracts

```
library Set {
  // We define a new struct datatype that will be used to
  // hold its data in the calling contract.
  struct Data { mapping(uint => bool) flags; }

  // Note that the first parameter is of type "storage
  // reference" and thus only its storage address and not
  // its contents is passed as part of the call.  This is a
  // special feature of library functions.  It is idiomatic
  // to call the first parameter 'self', if the function can
  // be seen as a method of that object.
  function insert(Data storage self, uint value)
      returns (bool)
  {
      if (self.flags[value])
          return false; // already there
      self.flags[value] = true;
      return true;
  }
}
```

```
contract C {
    Set.Data knownValues;

    function register(uint value) {
        // The library functions can be called without a
        // specific instance of the library, since the
        // "instance" will be the current contract.
        if (!Set.insert(knownValues, value))
            throw;
    }
}
```

Solidity Contract Inheritance

```
contract owned {
    function owned() { owner = msg.sender; }
    address owner;
}


// Use "is" to derive from another contract. Derived
// contracts can access all non-private members including
// internal functions and state variables. These cannot be
// accessed externally via `this`, though.
contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
}
```

Solidity Contract Inheritance

Abstract Contract

```
contract Config {
    function lookup(uint id) returns (address adr);
}
```

Multiple Inheritance

```
contract named is owned, mortal {
    function named(bytes32 name) {
    }
    // Functions can be overridden, both local and
    // message-based function calls take these overrides
    // into account.
    function kill() {
        super.kill();
    }
}
```

Solidity Contract Inheritance

Passing constructor arguments

```
contract Base {
    uint x;
    function Base(uint _x) { x = _x; }
}

contract Derived1 is Base(7) {
    function Derived1() {
    }
}

contract Derived2 is Base {
    function Derived2(uint _y) Base(_y * _y) {
    }
}
```

Standard Contracts

import "std"

```
contract owned {
  address owner;
  function owned() {
    owner = msg.sender;
  }
  function changeOwner(address newOwner) onlyowner {
    owner = newOwner;
  }
  modifier onlyowner() {
    if (msg.sender==owner) _
  }
}

contract mortal is owned {
  function kill() onlyowner {
    if (msg.sender == owner) suicide(owner);
  }
}
```

Standard Contract: Token

```
contract Token {
    function totalSupply() constant returns (uint256 supply) {}
    function balanceOf(address _owner) constant returns (uint256 balance) {}
    function transfer(address _to, uint256 _value) returns (bool success) {}
    function transferFrom(address _from, address _to, uint256 _value) returns (bool success) {}
    function approve(address _spender, uint256 _value) returns (bool success) {}
    function allowance(address _owner, address _spender) constant returns (uint256 remaining) {}

    event Transfer(address indexed _from, address indexed _to, uint256 _value);
    event Approval(address indexed _owner, address indexed _spender, uint256 _value);
}
```

Standard Contract: NameReg

```
contract NameReg {
  function register(bytes32 name) {}
  function unregister() {}
  function addressOf(bytes32 name) constant returns (address addr) {}
  function nameOf(address addr) constant returns (bytes32 name) {}
  function kill() {}
}
```

```
contract coin is named('MyCoin') {
  function coin(bytes3 name, uint denom) {

  }
}
```

Default ideas

Marketplace + Reputation + Trusted Identity Guard

Vickrey Auction

Word Guessing Game

Token Market

Multi Subscription Contract (Collect all payments)

Developing the contract

Testing

Mostly done in Solidity or JS

Many different approaches

All of them come with disadvantages

Testing with Solidity

Dapple

solUnit (no longer developed, successor?)

Testing with JS

Embark (with jasmine)

Truffle (and pudding) with Mocca Chai

Testing - Other

populus / py.test

JAVA and ethereumJ

ethereumjs-testrpc

node + ethereumjs-vm

Test Scenarios

All use cases

Unauthorised access

Users sends too few / too much

Different gas limits (to detect uncaught errors)

Use addresses with nonce 0

Use address of contract as argument

Call Recursion

Test Scenarios

Different gas limits (to detect uncaught errors)

```
contract token {
    mapping (address => uint) balances;
    function send (address a, uint256 v) {
        if(balances[a] >= v) {
            balances[a] -= v;
            a.send(v);
        }
    }
}
```

Test Scenarios

Addresses with nonce 0 (send might cost more gas)

```
contract token {
    mapping (address => uint) balances;
    function send (address a, uint256 v) {
        if(balances[a] >= v) {
            balances[a] -= v;
            a.send(v);
        }
    }
}
```

Test Scenarios

Call Recursion / Reentrance

```
contract token {
    mapping (address => uint) balances;
    function send (address a, uint256 v) {
        if(balances[a] >= v) {
            if(a.call.value(v)()) {
                balances[a] -= v;
            }
        }
    }
}
```

Test Scenarios

Call Recursion / Reentrance

During **any call** you contract can be called again (unless low gaslimit)

After **any call**, all assumptions about the state of the contract are invalid

Testing the contract

Deployment

Ethereum Studio

UI Wallet

geth

node.js

Deployment using UI

Demo

Deployment using geth

Demo

Deployment using node.js

Demo

1vieCmqYB3DE8StinXYBGGvgJ9hoXP1ib

The End

0x50008dd0cc879e0341042f97541eb4870c9c8393