Solidity 101 Workshop

presented by:

Agenda

- Ethereum
- Smart Contract
- Solidity Basics
- Additional Smart Contract Theory
- Escrow Contracts

What is Ethereum

- Open platform software based on blockchain technology
- Creation and deployment of **decentralized applications**
- Broader range of use cases as compared to Bitcoin (which is solely a cryptocurrency)
- Runs on **Ether**: a crypto token that fuels the network
- Main programming language: Solidity

Ethereum Virtual Machine

- A runtime environment for smart contracts
- Transaction based machine
- Takes advantage of a **consensus protocol** to make decisions (currently Proof of Work)

What is a Smart Contract?

- A **self-executing** program that automates the **verification of conditions**
- Allows credible transactions **without 3rd parties**
- **Trackable, immutable** (stored on blockchain)

Smart Contract In Practice

- A container for information (**state variables**) and functionality (**functions**)
- **Data is stored** on the contract, and execution happens when conditions are met
- Contracts **cost GAS to run**, which is paid for with ether

Polymorphism

- Smart Contracts don’t have to be independent
- Import statements and class extension for inheritance
- Syntax:

```
import "./animal.sol";

contract Dog is Animal{}
```

Smart Contract: Use Cases

- **EtherDelta**
  - Implements an exchange to exchange of Ether
  - Checks for sufficient funds (tokens) for both sides of the deal and runs the exchange.
  - Replaces middleman such as a bank!

- **Marriage Contract**
  - One party proposes to second party.
  - If both parties agree, funds are withdrawn from both parties' accounts and put into joint account.
  - Status changed to "married"
  - If one party requires "divorce", funds are split into two halves!

- **Etheroll**
  - Gambling on the EVM!
  - Place bets and profit is transferred to your account when won
  - Comparable to casino!

Solidity Basics

To start off

Compiler Version

- “^” limits compiler version up to 0.5.0

```
pragma solidity ^0.4.24;
```

Declare Contract

- The start of every contract

```
contract Blockchain{

}
```

Variable Types

- **boolean**: true, false
- **int/uint**: signed and unsigned integers
- **string**: dynamically-sized string, e.g.: “Welcome to Solidity 101”
- **enum:** user-defined type

- **address**: an Ethereum address. 20 bytes

  (more on this later)

Operators

- **Comparison** (boolean)
  - >= , > , <= , < , != , ==, &&
- **Arithmetic** (uint)
  - +, -, *, **, /, %

Reference Types

Arrays

- Used when you want a collection of a specific type.
- Can be of 2 types:
  - **Fixed:** has a fixed length of elements.
  - **Dynamic:** no fixed size; can keep growing

```
uint[] dynamicArray;
string[5] fixedStringArray;
```

Structs

- A custom type with multiple fields/variables

```
struct myself {
    string name;
    uint age;
    bool alive;
}
```

Addresses

- The Ethereum network is made of **addresses** that are owned by **users** or **smart contracts** which can store Ether, similarly to personal bank accounts

- An address is 160 bits (base 2) long, or 40 hex digits (base 16) They look like this : `0x0cE446255506E92DF41614C46F1d6df9Cc969183`

Mapping

- Storage of key-value pairs *(think of Hash Tables)*

Example use of Addresses and Mapping

```
mapping (address => uint) favoriteNumber;

// Set value of favorite number at sender address
function setMyNumber(uint _myNumber) public {
  favoriteNumber[msg.sender] = _myNumber;
}

// Return value of sender address
function whatIsMyNumber() public view returns (uint) {
  return favoriteNumber[msg.sender];
}
```

Functions

- ```
  function average (uint a, uint b) public returns (uint) {
      returns (a+b)/2
  }
  ```
- Similar to other object-oriented programming languages
- General syntax of a function:

```
function NAME() VISIBILITY MODIFIER returns (RETURN TYPE) { }
```

- Syntax of a function for calculating average value:

```
function average (uint a, uint b) public returns (uint) {
        returns (a+b)/2
{
```
```

Functions cont.

Function Visibility

Remember functions produce actions BUT you don't always want everyone to be able to access them

- **external**: accessible from the outside only, part of contract interface
- **public**: accessible from everywhere, part of contract interface (functions are public by default)

- **internal**: only accessible internally, and in derived contracts
- **private**: only accessible inside the contract it is defined in (but not derived contracts)

Functions cont.

Function Modifiers

- **view:** does not modify data, only views it

```
function myFunction() public view returns(uint) { }
```

- **pure:** does not access contract data, only parameters

```
function multiply(uint a, uint b) private pure
returns(uint) { return a * b; }
```

Error Handling/ Conditions

Require

- The require function is meant to be used for **input validation.**
- Requires certain condition to **continue** with the code; otherwise

  reverts and refunds gas if the condition is false.

```
function setUserAge(uint age) public {
    require(age > 0);
    Myage = age;
}
```

Can be a great way to make sure only a certain user can call a function (if you would like to limit the function's use)

Error Handling/Conditions cont.

Assert

- Also reverts when the condition is false, but uses all gas
- Often used for internal errors or to check for invalid state to detect bugs

```
contract Sharer{

    uint balance = 100;

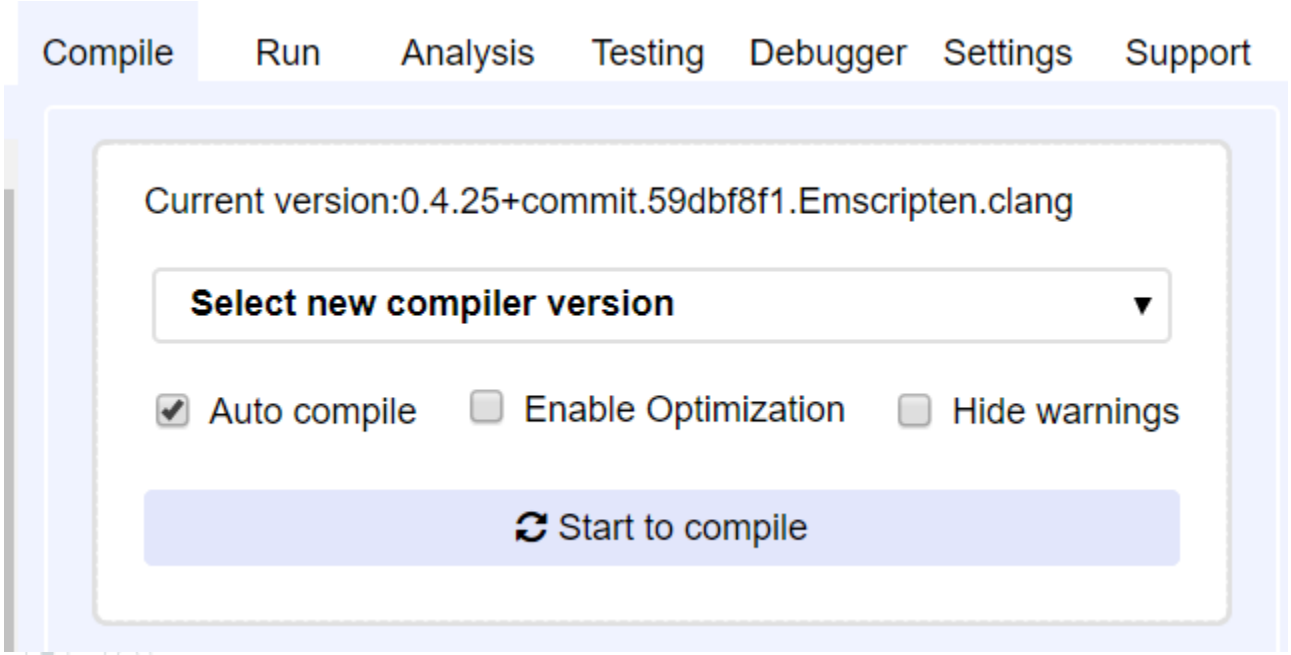
    function sendHalf (address addr) public payable returns (uint _balance){
        //Ensures there is value in account
        require (msg.value>0);
        uint balanceBeforeTransfer = this.balance;
        addr.transfer(msg.value/2);
        //Checks for the balance is actually halved
        assert(this.balance == balanceBeforeTransfer-msg.value/2);
        return this.balance;
    }
}
```

Your Turn!

Make a Hello World contract

Setting up Remix

- Open remix.ethereum.org
- Select Auto compile



Remember

1. Compiler version
2. Declare Contract
3. State variables
4. Functions
5. Return output!

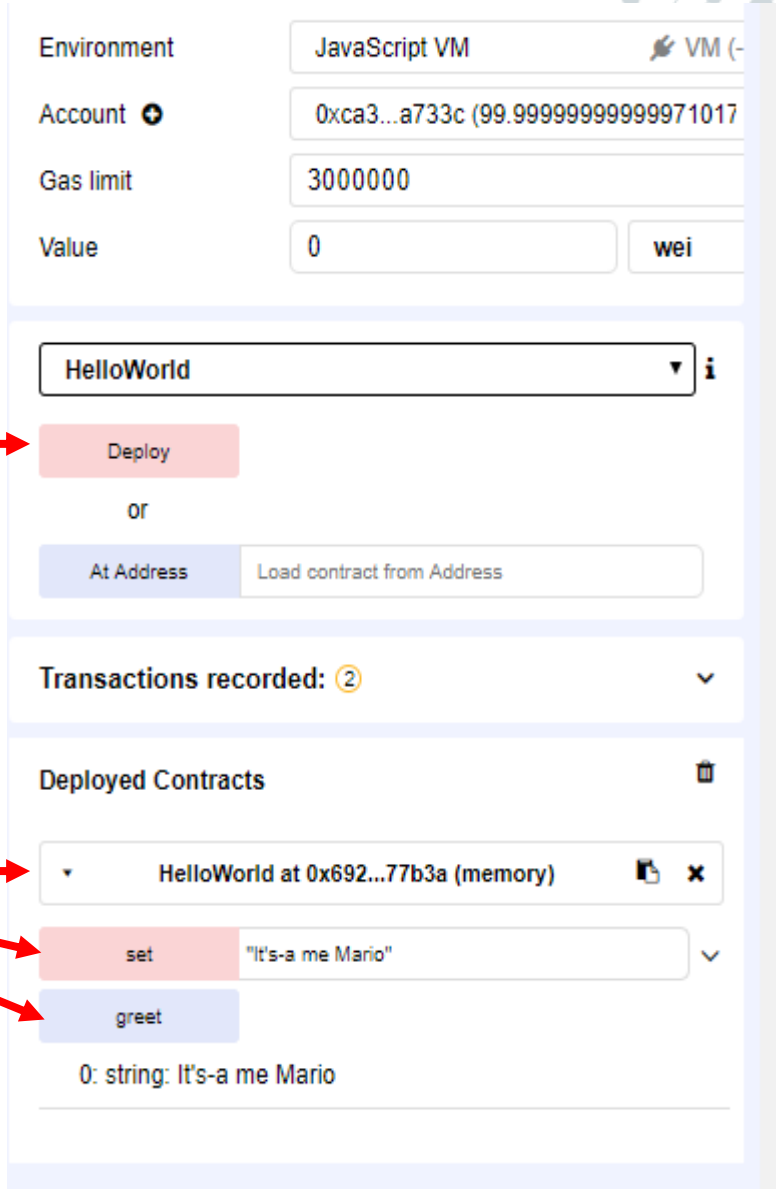
Hello World Pseudocode

```
1   //declare compiler version
2   .
3   .
4   .
5
6   //declare contract
7
8       //State variable that will be used in the contract, stores data permanently
9       .
10      .
11
12
13      //Declare function that takes string and sets to declared variable (default is public)
14      .
15      .
16
17      //Public function returning string, a view function means that
18      //the function does not modify the state of the contract
19      .
20      .
21      .
```

Try it!

Deploying the Contract

Click “Deploy”



Click the dropdown. Type your greeting with quotation marks and click “set”, then click “greet”

Questions so far?

Block and Transaction Properties

- **msg.value** : value (wei) of message sent during function call
  - i.e.: require(msg.value == 25);

- **msg.sender** : address of contract initializer or function caller

More Solidity Theory

Payable functions

- Allows contracts to receive and hold funds

```
function deposit() public payable {
    deposits [msg.sender] += msg.value
}
```

- Functions can take in a certain amount of Ether in **wei**
  (1 wei = 1e-18 ether)

Note: Without the "payable" function keyword, any transactions involving payments will be reversed!

Gas on Ethereum Network

Gas concept

- Every operation that can be performed by a transaction or contract costs a certain amount of gas which is paid in ether

- Cost is proportional to **computation cost**

- **The market decides the most viable ether/gas cost proportion**

*Distinction: gas **cost** is the amount of work required to perform an action in the EVM, gas **price** is the amount of ether you pay*

Gas on Ethereum Network

Gas economics

- If your gas *price* is too low, no one will process your transaction
- If your gas *price* is high, your transaction will be run but you will pay more than you might have needed to.
- If your gas *price* is fine but the gas *cost* of your transaction runs "over budget" the transaction fails but still goes into the blockchain, and **you don't get the money back** for the work that the miners did.
- **This makes sure that nothing runs forever, and that people will be careful about the code that they run. It keeps both miners and users safe from bad code!**

Storage vs Memory

- **“Storage”** is where all the contract’s state variable reside.
  - Every contract has its own storage and will **remain** there between function calls.
  - It is quite **expensive** to use since it will be using storage space in the EVM
  - Is the default setting.

- **“Memory”** is used to hold **temporary** values.
  - It is erased between external functions calls.
  - Much **cheaper** to use since it will not use storage space in the EVM.
  - Ideal to use for intermediate variables inside functions.

Cryptographic Hash Functions

- Used to map data of arbitrary size to data of a fixed size.
- Returns hash values or hash codes.
- Meant to encrypt data securely
- Ethereum uses “**keccak256**”

1. They are **irreversible**
2. **Unique**; 2 different data cannot have same hash.
3. “**Avalanche effect**”; any small change results in big change of hash.
4. **Deterministic**; same input always has same hash.

Escrow Contract

- A contract that serves as the third person (**escrow agent**) in a contract
- The code imitates the legal document that outlines the **terms and conditions** agreed between parties
- Ensures both parties **fulfill its obligations**
- The buyer deposits assets while the seller delivers the goods who will then receive the assets once the buyer receive the goods

Writing an Escrow Contract!

Step one:

```
1  //declare solidity version - we will be using 0.4.24
2
3  //declare Escrow contract where we will start by stating the variables
4    //variables used will be:
5    //a uint representing a balance
6    //a public address for the buyer
7    //a public address for the seller
8    //a boolean that will represent the agreement (or not) of the buyer
9    //a boolean that will represent the agreement (or not) of the seller
10   //a private uint that will keep track of time
```

Writing an Escrow Contract!

Step two:

```
12  //define a constructor function that will only run once upon
13  //initialization. You can name it the same a the contract
14  //this function will take two addresses (buyer and seller) and should be a
15  //public and payable function
16    //set one of the inputed addresses to one of the stated variables
17    //in the contract (buyer or seller)
18    //repeat for the second inputed addresse with the other stated variable
19    //also set the time tracker to "now", which represents the moment
20    //that the fucntion is called.
21
```

Writing an Escrow Contract!

Step three:

```
22    //define a function (accept) that will do 2 things: allow both the buyer
23    //and seller to accept the transaction and execute the transaction
24    //or, if one of them doesnt accept, reverse the transaction
25    //this function wont take any variables and should be public
26      //first, if who called the function is the buyer,
27          //set the buyer agreement to "true".
28          //msg.sender gives you the address of who is calling
29        //if the caller isn't the buyer, check that it is the seller and
30          //set his agreement to "true"
31      //next, if both agreed, execute the transaction using another function
32          //that we will later define as "payBalance()"
33        //if the agreements are different and "now" is bigger than our
34        //tracker + 30 days,
35          //selfdestruct(buyer)
36          //this will allow for a 30 day period to "change their mind"
37
```

Writing an Escrow Contract!

Step four:

```
38    //define a function to pay the seller
39    //call it payBalance(), this function also wont take any
40    //variables and should be private
41      //"seller.send(_)" is a good tool
42      //this.balance will give the value of the balance
43
44    //define a function for the buyer to deposit his "money"
45    //name it "deposit()". It also wont take any variables and
46    //should be public and payable
47      //check that the funciton caller is the buyer and
48        //increase the stated variable for balance with the value of the
49        //message using msg.value
```

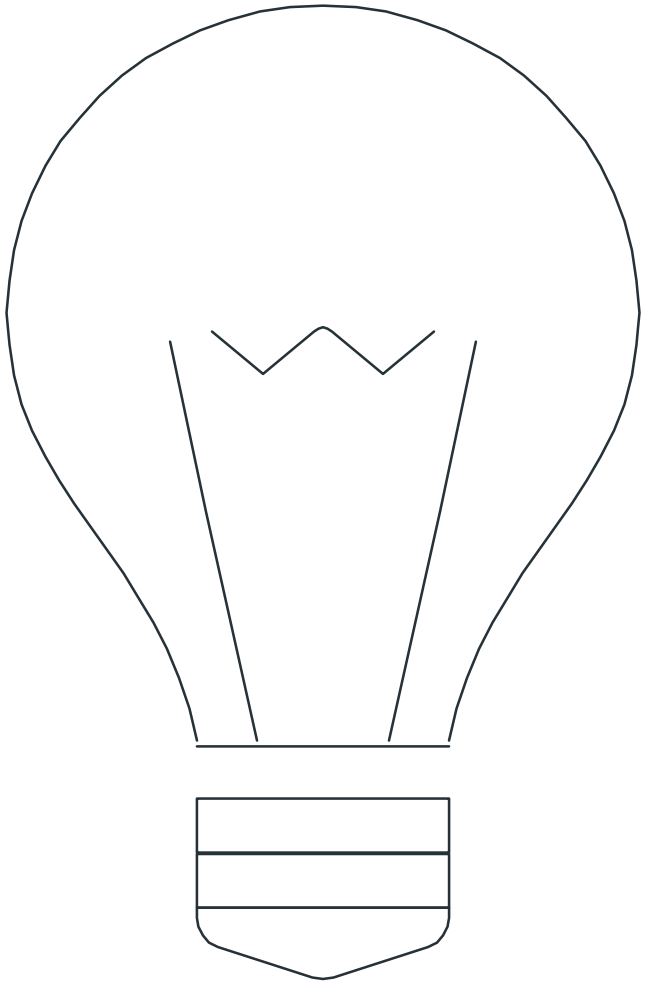
Writing an Escrow Contract!

Step five:

```
51    //define a function that will allow either buyer or seller
52    //to cancel transaction. name it "cancel()" and should be public
53      //if function caller is buyer
54          //set buyer agreement to "true"
55        //if function caller is seller,
56          //set seller agreement to "true"
57  //the reason we set them to "true" and not "false" is because false is
58  //the default value of boolean in solidity. because of the rest of the code,
59  //working with false would not work.
60      //if both agreements are set to "true", return funds to buyer using
61      //selfdestruct(buyer)
62
```

Let’s go over it!

Final Questions?



Other basics to explore

- Fallback functions
- Abstract contracts and interfaces
- Other block and transaction properties
- Events and javascript implementations
- Truffle and Web3.js implementations

Conclusion & More

What we taught you today

There’s still a lot left to learn, but don’t be intimidated!
Here are resources you can use:

Ethereum White Paper:
https://github.com/ethereum/wiki/wiki/White-Paper

Solidity Development Documentation:
https://solidity.readthedocs.io/en/develop/

Thank you for coming!

Look forward to future
Blockchain at McGill events!