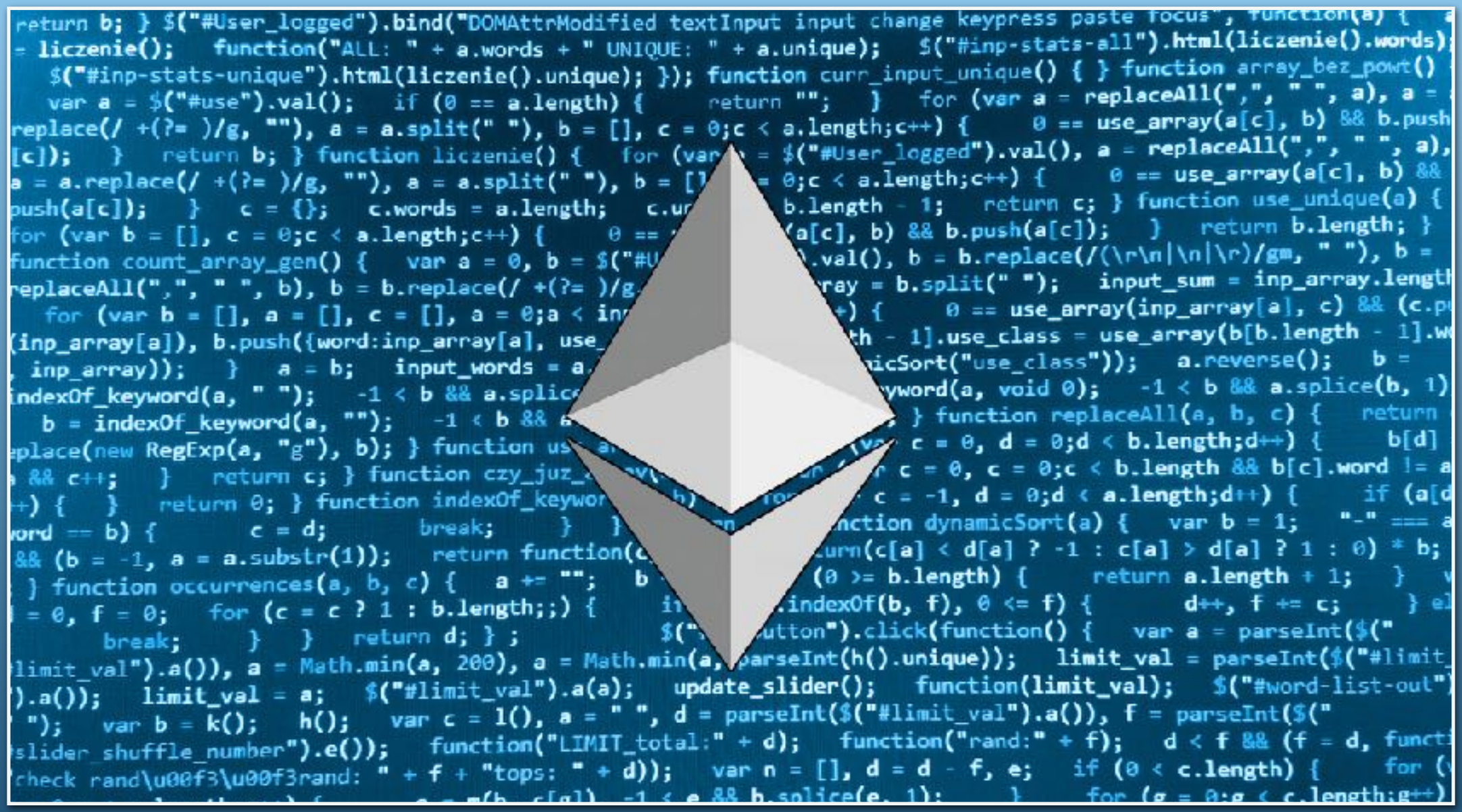

Solidity Basics

A quick understanding of Solidity by Example

Experience is the best teacher…

Find a Problem to Solve.

Find examples.

Build it.

Integrate Ethereum Payments With This Robot

https://www.wired.com/video/the-robot-that-s-roaming-san-francisco-s-streets-to-deliver-food

Problem Defined

A customer orders Mediterranean food and wants to pay with Ether. Both parties have Ethereum accounts setup.

We need to build a Smart Contract that the Restaurant will own and where customers can place their orders.

Problem Defined

Customers should be able to pay for their order through the contract, and be able to cancel the order any time before delivery.

Once the code is given to the robot, the robot releases the food while at the same time the money is released to the restaurant.

Learning By Example

https://solidity.readthedocs.io/en/latest/solidity-by-example.html

Purchase Contract

```
// Simple Contract based on
// Safe Remote Purchase example
// https://solidity.readthedocs.io/en/latest/solidity-by-
example.html#safe-remote-purchase

pragma solidity ^0.4.0;

contract Purchase {
    uint public value;
    address public seller;
    address public buyer;
    enum State { Created, Locked, Inactive }
    State public state;

    function Purchase() payable {
        seller = msg.sender;
        value = msg.value / 2;
        if (2 * value != msg.value) throw;
    }
```

Some Solidity Data Types

- Booleans - denoted by bool with values of true, false
- Integers - int, uint aliases of uint256; int8-int256 step 8
- Addresses - 20 bytes; 0x7898… Ethereum Address
- Bytes & Strings - bytes1 .. bytes32; string is UTF-8
- Arrays - fixed or dynamic; a[0]; a.length; a.push;
- Enums - Way to create user-defined types. Uses ints
- Structs - Way to build objects. struct a { uint amt; }
- Mappings - Way to build hash tables.

Solidity Keywords

- Public - Makes the variable readable by the outside world. It generates an accessor function when compiled.
- Payable - Used in reference to a function that allows payment to be made to the contract.
- msg.sender - address of the one calling the function.
- msg.value - value in Wei sent to the function
- Constant - Declares that the function does not change state.
- Internal / External - Functions default to internal. External is the way to communicate outside the contract.
- Throw - Terminates the transaction and reverts (returns gas)

Solidity Keywords

- this - Resolves to the current Ethereum address of the contract. It is an address type.
- <address>.send(uint amount) - Sends an amount of Wei to an address. Returns false on failure.
- <address>.transfer(uint amount) - Send an amount of Wei to an address. Throws on failure.
- <address>.balance - Returns the balance of an address. this.balance is most commonly used.
- <address>.call(…) - Calls a function from another contract. Return data from the other contract is not available.
- <address>.delegatecall(…) - Calls the function within the current contract environment (storage, balance, state, etc…) Its purpose is to use library code which is stored in another contract.

Contract Initiation

- Init Function - Contracts use a function with the same name as the contract to pass and initialize variables.

```
contract Purchase {

    // By having the init function ‘payable’,
    // you can add ether to the contract when it is
    // deployed to the blockchain.

    function Purchase() payable {
        // CODE HERE ONLY RUNS ONCE.
        seller = msg.sender;
        value = msg.value / 2;
        if (2 * value != msg.value) throw;
    }

    …
}
```

Purchase Contract

```
modifier require(bool _condition) {
    if (!_condition) throw;
    _;
}

modifier onlyBuyer() {
    if (msg.sender != buyer) throw;
    _;
}

modifier onlySeller() {
    if (msg.sender != seller) throw;
    _;
}

modifier inState(State _state) {
    if (state != _state) throw;
    _;
}
```

Modifiers

- Modifier - A code segment that prepends to a function call. It uses the special variable _;

```
modifier onlySeller() {
    if (msg.sender != seller) throw; _;
}

 function abort() onlySeller {
      aborted();
      …
}
```

```
 function abort() {
      if (msg.sender != seller) throw;
      aborted();
      …
}
```

Purchase Contract

```
    event aborted();
    event purchaseConfirmed();
    event itemReceived();

    /// Abort the purchase and reclaim the ether.
    /// Can only be called by the seller before
    /// the contract is locked.
    function abort()
        onlySeller
        inState(State.Created)
    {
        aborted();
        state = State.Inactive;
        if (!seller.send(this.balance))
            throw;
    }
```

Events

- Event - Works with Javascript listeners to trigger external code.

```
event aborted(string message);

function abort() onlySeller {
      aborted(“Seller Aborted”);
      …
  }
```

```
// JAVASCRIPT Code
// How to add a listener for the Purchase.aborted EVENT
var listen = Purchase.aborted( {}, function(err, res) {
  if (!err) {
    var msg = “Purchase Event: ”+res.args.message;
    console.log(msg);
 }
});
```

Enums

- Enums - These are user-defined types often used to define the current contract state. The actual values of these fields resolve to integers starting at 0 .. x

```
enum State { Created, Locked, Inactive }

// State.Created always equals 0;
// State.Locked always equals 1;

function abort()
    onlySeller
    inState(State.Created)
{
    aborted();
    state = State.Inactive; // This equals 3
    if (!seller.send(this.balance))
        throw;
}
```

Purchase Contract

```
/// Confirm the purchase as buyer.
/// Transaction has to include `2 * value` ether.
/// The ether will be locked until confirmReceived
/// is called.
function confirmPurchase()
    inState(State.Created)
    require(msg.value == 2 * value)
    payable
{
    purchaseConfirmed();
    buyer = msg.sender;
    state = State.Locked;
}
```

Purchase Contract

```
    /// Confirm that you (the buyer) received the item.
    /// This will release the locked ether.
    function confirmReceived()
        onlyBuyer
        inState(State.Locked)
    {
        itemReceived();
        // It is important to change the state first
        // because otherwise, the contracts called
        // using `send` below can call in again here.

        state = State.Inactive;
        // This actually allows both the buyer and
        // the seller to block the refund.
        if (!buyer.send(value) ||
             !seller.send(this.balance))
            throw;
    }
}
```

Sending Value

- Enums - These are user-defined types often used to define the current contract state. The actual values of these fields resolve to integers starting at 0 .. x

```
    enum State { Created, Locked, Inactive }

    // State.Created always equals 0;
    // State.Locked always equals 1;

    function abort()
        onlySeller
        inState(State.Created)
    {
        aborted();
        state = State.Inactive; // This equals 3
        if (!seller.send(this.balance))
            throw;
    }
```

Additional Concepts

Owned & Mortal Contracts

```
contract owned {
    address public owner;

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

    modifier onlyOwner {
        if (msg.sender != owner) throw;
        _;
    }

}

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

Two Data Structures

- Arrays & Mappings - These are two common ways of storing data.

```
uint public purchaseCount;
struct Purchase {
    uint value;
    address buyer;
}
//Creates a mapping of Campaign datatypes
mapping (uint => Purchase) public purchases;
mapping (address => uint) public purchase_hash;
Purchase[] public purchase_array;
```

Two Data Structures

- Arrays & Mappings - These are two common ways of storing data.

```
function add_purchase(uint _amount) {
    // Adding a new investor record
    Purchase p = purchases[purchaseCount];
    p.buyer = msg.sender;
    p.value += _amount;
    purchase_hash[msg.sender] = purchaseCount;
    purchaseCount++;

    purchase_array.push(p);
}

function update_purchase(address _user, uint _amount) {
    uint purchase_id = purchase_hash[_user];
    purchases[purchase_id].value = _amount;

    // purchase_array[??]
}
```

Now Let’s Build a RoboPurchase Contract