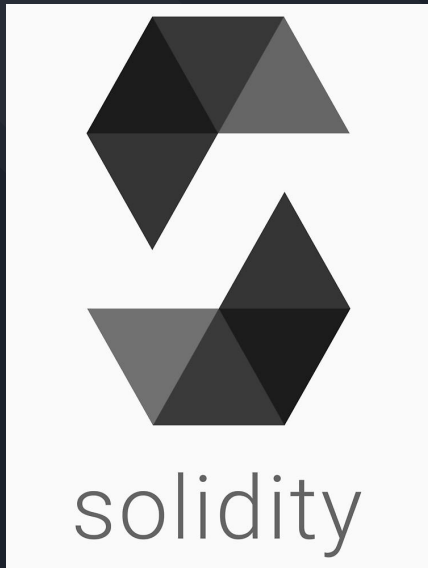


Smart Contract Creation



Brendan Furtado



Overview

- Solidity Review & a few coding examples
- Creating a Smart Contract for a Shop (problem to code)
 - Walkthrough



Solidity

- Programming Language: Solidity
 - Solidity is an object-oriented, high-level language for implementing smart contracts.
 - Solidity was influenced by C++, Python and JavaScript and is designed to target the Ethereum Virtual Machine (EVM)
 - The EVM can read and execute smart contracts
 - Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features.



Gas? Ether?

- Gas is a special unit used in Ethereum to measure the work an action takes to perform
- Every operation that can be performed by a transaction or contract on the Ethereum platform costs a certain number of gas
 - Important because it helps ensure an appropriate fee is being paid by transactions submitted to a network
 - Helpful to distinguish between the price of computation from a fixed price, so it doesn't change every time market shifts up and down
- Ether is the token that powers the network built on the Ethereum platform
- Wei is a denomination of ether $1 \text{ ether} = 10^{18} \text{ wei}$



Learning to code in Solidity

- Overview of basic Solidity
 - Some coding examples
- Before jumping into Shop creation
 - Break down the problem (looking for basic functionality):
 - What functions are needed to create a shop?
 - What information is useful to provide?
 - (more complicated) who can access that information
 - What logic is needed to create those functions
- Will try to answer questions
 - Logic, Syntax, etc.



Overview 1: Value Type Variables

- Booleans denoted: bool (true/false)
- Integers: int (signed integer), uint (unsigned integer)
 - Can be set up in steps of 8 (example: uint8, 8 bits) up to 256 bits
- String: string varName
 - Max length 32-bytes (256-bit)
- Address denoted by keyword: address
 - Holds a 20 byte value for an ethereum address
 - Members: balance, transfer



Value Type Declaration

- `uint public number;`
- `bool public b;`
- `string name = 'Bob';` //Assigning a fix value
- `address public contractOwner;`
 - `contractOwner` set by constructor (later)



Overview 2: Special Variables

- There are special global variables which always exist in the global namespace and are used to provide information about the blockchain
- Block and message object
- Message main focus
 - msg.value: is a member of the message object when sending transactions on the Ethereum network
 - Contains the amount of wei (1 ether = 10E18 wei) sent in the transaction
 - msg.sender: sender of the message (the current call), i.e the address where the current function call came from
 - Use within functions, can set addresses to them, etc.



Storage and Memory keywords

- Definition: **State variables** (think of them as internal contract variables that can be used among the contract's functions), they are variables whose values are permanently stored in contract storage.
 - Similar to characteristics/property variables of a java class
- Types of storing: storage & memory
- **storage**: where all state variables reside
 - Persistent between function calls and expensive to use (permanent space to takes up more memory)
- **memory**: used to hold temporary values
 - Erased between (external function) calls and cheaper (less space taken up)
- Also one more called Stack, not as important, but it basically holds small local variables



State/Global Variable Location

- Within the contract barriers usually at the top

```
pragma solidity >=0.4.16 <0.6.0;  
  
contract Simple {  
  
    uint public a = 10;  
    address me;  
  
    //Below are written functions and other code  
  
}
```



Msg.value & msg.sender & memory

```
function randomFunction(uint ID){  
  
    Item memory item = items[ID];  
  
    require(item.owner == msg.sender);  
  
    require(msg.value == item.price);  
  
}
```

- Requires the item's owner to be the address of whoever is calling the function, msg.sender
- Requires that whatever amount of wei being sent in the function is (msg.value), it has to be equal to the item's price
- Failure otherwise



Overview 3: Reference Types - Array

- Reference types are: Arrays, Mappings and Structs
- Arrays can have a compile-time fixed size, or they can have a dynamic size
 - To be dynamic, array must be in storage, aka it must be a state variable
 - Ex. Contract example{ uint[] myArray; function(){ can access and push to array}}
 - For fixed size, the memory cannot be resized
 - Declare with memory keyword within a function
 - Ex. uint[] memory myArray = new uint[](7);
 - Length placed in parenthesis



Side Note: Add & Delete elements in an array

- When you have a global array variable (state variable) that is a dynamic array, use:
 - `arrayName.push(valueType)` that appends an element to the end of the array
- When you have a local array variable (memory):
 - Can directly set an index to a value Type
 - `uint[] memory arr = new uint[](5);`
 - `arr[0] = 10;`
- Simple deletion: `delete array[index];` (leaves gap)
 - Other ways: Swap & Delete, Delete & Shift

Overview 3: Reference Types - Mapping

- Mappings act as hash tables, consists of key types and corresponding value type pairs
 - `mapping(keyType => valueType) public mappingName;`
- Used a lot in associating Ethereum addresses with a value type
 - `mapping(address => uint) public userLevel;`
- Usually state variables
- Ex. a bunch of users (addresses) associated with a level

address (key)	→	userLevel (value)
0xDfc	→	2
0xL3g	→	3
0xZ1b	→	1

Multiple users can play, each having
a corresponding level assigned

keys	hash f()	index	value
_key1	→	0	_value6
_key2	→	1	_value3
_key3	→	2	_value5
_key4	→	3	_value4
_key5	→	4	_value2
_key6	→	5	_value1



Accessing Elements in a Mapping

- Example of a Mapping variable:
 - `mapping (address => uint[]) public stuffOwned;`
 - Within a function you can write: `stuffOwned[msg.sender].push(ID);`
 - Which adds a number to that list
- Breaking it down
 - `stuffOwned[msg.sender]`
 - Whoever the caller is, the address of that caller is directly connected to a distinct array of integers
 - That array of integers is what the caller/address owns
 - `.push(ID)` pushes/adds to that array above
 - In this case it is an array of IDs that represents what the address owns



Overview 3: Reference Types - Structs

- Same thing seen in C programming
- Basically a user defined data type
 - Structs can have their own characteristics/variables
- Example
 - It's defined within a contract
 - You can now create person “objects” or types within functions or as global variables

```
contract structExample{  
  
    struct person{  
        string name;  
        string homeAddress;  
        uint phoneNumber;  
    }  
}
```




Overview 4: Functions part 1

- Function format:

```
function name([argument, ...]) [visibility] [view|pure] [payable] [custom modifier, ...] [returns([argument, ...])] {
```

```
    Function body;
```

```
    return(); //return statement depending on function
```

```
} //End of function
```



Functions part 2: Visibility

- external: Part of the contract interface, which means they can be called from other contracts and via transactions,
 - Cannot be called internally
- public: Can be either called internally or externally through message calls
 - Side: public variables have automatic getter functions (access to variable's data)
- internal: Functions and state variables can only be accessed internally (from within the current contract)
- private: Only visible for the contracts they are defined for and not in derived contracts (inheritance)
 - Prevents other contracts from accessing and modifying the information



Functions part 3: Modifiers

- Conveniently modify function behavior
- View: Promises not to alter or modify the state of the blockchain but can read information from anywhere
 - Cannot send or receive ether
- Pure: View functions with more restrictions, only concerned with data that is passed or in scope
 - Not allowed to read or modify, also cannot send or receive ether
- Constant: Recently deprecated to view and pure, view generally replaces it
- Payable: Allows a function to receiver ether while being called
 - All ether sent to payable functions are owned by the contract



Custom Modifier

- Can have custom modifier like

```
modifier onlyOwner () {  
    require(msg.sender == owner);  
    _;  
}
```



Functions part 4: Return Statements

- If you want your function to return a value use the returns keyword
- The returns keyword is followed by (V0, V1)
- Format:
 - function nameOfFunction(type param1, type param2) visibility modifier returns (type name){
//function body... return (type) }

```
pragma solidity >=0.4.16 <0.6.0;  
  
contract Simple {  
  
    function arithmetic(uint _a, uint _b) public pure returns (uint o_sum, uint o_product){  
        return (_a + _b, _a * _b);  
    }  
}
```

- Number of components must be the same as the number of return types



Require statements

- Like a conditional check
- Throws an exception when a condition is not met
- Example:

```
modifier onlyOwner () {  
    require(msg.sender == owner);  
    _;  
}
```



Functions part 5: Modifier Payable

- With payable functions, you can send and receive ether, how can you do this?
 - `address.send(amount)`
 - `address.transfer(amount)`
- Both send and transfer use a limit of 2300 gas as a fixed payment
 - Difference is, send doesn't throw an exception when gas runs out/exceeds limit
 - Fix with a `require` statement
 - transfer throws an exception when performed unsuccessfully, you know your transaction is unsuccessful right after execution



Functions part 6: Constructors

- When contract is deployed/created, it calls the constructor
- Constructors initialize starting values to whatever the user wants
- Constructor functions can be either public or internal (visibility)
- If no constructor, the contract will assume a default constructor
- If you're sending value (in ether or wei)
 - Constructor must be payable

```
constructor() public payable{  
    owner = msg.sender;  
}
```

```
pragma solidity >0.4.99 <0.6.0;  
  
contract A {  
    uint public a;  
  
    constructor(uint _a) internal {  
        a = _a;  
    }  
}
```




Coding Examples



```
pragma solidity >=0.4.0 <0.6.0;
```

```
contract SimpleStorage {  
    uint storedData;  
  
    function set(uint x) public {  
        storedData = x;  
    }  
  
    function get() public view returns (uint) {  
        return storedData;  
    }  
}
```

```
pragma solidity ^0.4.24;

contract addressExample {

    address owner;
    Item[] items;

    struct Item{
        string name;
        uint ID;
    }

    //Assume 3 items are in the array called "items";
    // items = [Book, Phone, Computer] at indexes 0, 1, 2 respectively
    //We call deleteItem on ID/Index 0.
    //What happens?
    function deleteItem(uint ID) public{

        Item memory item = items[ID];

        require(item.ID >= 0);

        if(items.length > 1){
            items[ID] = items[items.length - 1]; //Note: items.length - 1
                                                    //represents the last index
                                                    //of the array
        }
        items.length--;
    }
}
```



Walkthrough: Shop Smart Contract

- Problem: We want to create basic interaction of a shop
- What kind of functions are we gonna need?
- Any modifiers? Structs?
- How about state and global variables?
- Solidity reference guide: https://topmonks.github.io/solidity_quick_ref/



What we'll need

- Some global/state variables, address, array, and mappings
- Struct for an Item
- A function that lists an item
- Buy whatever is shown in shop
- A delete function
- Withdraw funds function
- Optional get items owned



UML Diagram & Coding

Shop.sol
<pre>struct Item +address contractOwner; +uint itemCount; +mapping(uint => Item) itemsOwned; +mapping(address => uint[]) itemsOwned; +mapping(address => uint) balances;</pre>
<pre>modifier onlyOwner() +constructor() payable +listItem(string name, uint price) onlyOwner +deleteItem(uint ID) +buyItem(uint ID) payable +withdrawFunds() +getItemsOwned() view: returns(uint[])</pre>



Item Struct

```
struct Item {  
    string name;  
    uint ID;  
    uint price;  
    address owner;  
}
```



Setup

- First initial set up on Remix, getting the compiler and contract ready
- From there, we will go step by step through the contract
- Tackle one or several tasks/functions at a time
- <https://remix.ethereum.org/#optimize=false&version=soljson-v0.4.25+commit.59dbf8f1.js>



Coding 1: Global Variables, Struct

5 Minutes



Coding 2: Modifier & Constructor

5 Minutes



Coding 3: listItem function

```
function listItem(string name, uint price) public onlyOwner {  
  
    //ID should be equal to the item count  
  
    //Increase count  
  
    //Use ID as an index for the mapping items, and create the object  
    you are trying to list  
  
    //Set the owner with the itemsOwned mapping  
  
}
```



Coding 4: buyItem function

```
function buyItem(uint ID) public payable{  
    //Create a local item object  
  
    //Require that the address calling the function is not buying what it  
    already owns  
  
    //Make sure you are sending the exact amount the item costs  
  
    //Increase seller's balance  
  
    //Change owner and add to buyer's list  
  
}
```



Coding 5: deleteItem function

```
function deleteItem(uint ID) public{  
    //Make sure that whatever you're deleting is owned by the caller  
    //Use delete keyword of items array at the correct index  
    //Decrease item count  
  
}
```



Coding 6: withdrawFunds function



Coding 7: getItemsOwned function

5 Minutes



Running on Remix!



Events

- Each transaction has an attached receipt which contains zero or more log entries, these entries represent the result of events having fired from a smart contract
- Events and logs are important because they facilitate communication between contracts and their user interfaces
 - When a transaction is mined, smart contracts can emit events and write logs to the blockchain that the frontend can then process
- You call events in the body of the functions you want to generate the event from



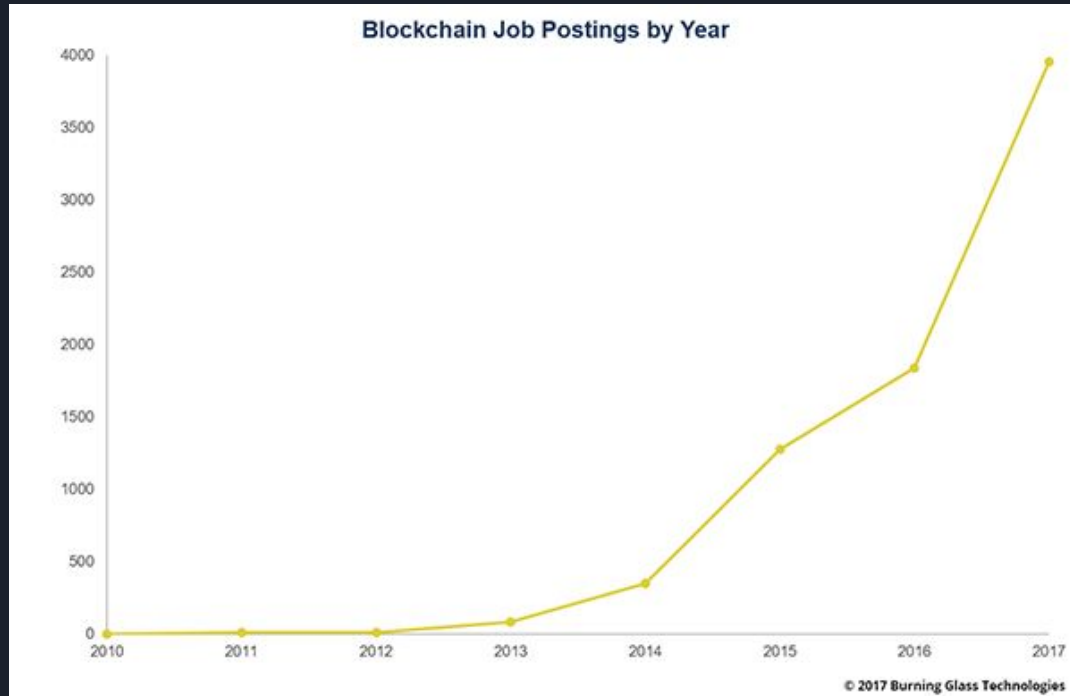
Concluding Remarks

- Practice!
 - Tutorials online
 - Youtube videos
 - <https://solidity.readthedocs.io/en/develop/types.html>
- Writing Solidity...
 - Takes time
 - Is lucrative
 - Ethereum developers, writing in Solidity can make >\$100,000
 - Smart Contract writers, >\$120,000

Sources:

<https://www.forbes.com/sites/shermanlee/2018/04/11/the-demand-for-blockchain-engineers-is-skyrocketing-but-blockchain-itself-is-redefining-how-theyre-employed/#404c2c916715https://hackernoon.com/blockchain-jobs-and-salaries-2018-report-45d3e7741c19>

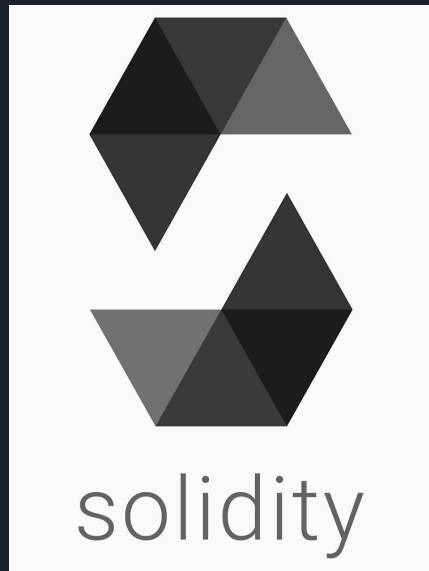
Increasing Demand




<https://hackernoon.com/blockchain-jobs-and-salaries-2018-report-45d3e7741c19>



The End





<https://medium.com/@ChainTrade/10-advantages-of-using-smart-contracts-bc29c508691a>

<https://hackernoon.com/blockchain-jobs-and-salaries-2018-report-45d3e7741c19>

<https://www.forbes.com/sites/shermanlee/2018/04/11/the-demand-for-blockchain-engineers-is-skyrocketing-but-blockchain-itself-is-redefining-how-theyre-employed/#404c2c916715><https://hackernoon.com/blockchain-jobs-and-salaries-2018-report-45d3e7741c19>

<https://medium.com/@davekai/how-hard-is-it-to-become-a-smart-contract-developer-f159bafd8018>

<https://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html#>

https://media.consensys.net/technical-introduction-to-events-and-logs-in-ethereum-a074d65dd61e?fbclid=IwAR3V2yqgM8AoPY4mMh80wfIO3hcSi8iiarvKQh0jFs6SmP_L4gbnFy6bKBc

Stack Exchange for Ethereum

Youtube Channels: https://www.youtube.com/watch?v=v_hU0jPtLto

https://www.youtube.com/watch?v=KU6bvciWgRE&list=PL0INJEnwfVVMuX2Ds19Wj_7Mcze3FDJr3