Vanbex Academy

Deeper dive into Solidity

The Solidity Programming Language

- Higher level language used to create smart contracts
- Looks like JS
- Contracts as Classes (OOP/COP)
- Statically Typed
- Supports Inheritance
- Compiles to bytecode

Structure

```
pragma solidity ^0.4.15;          // Compiler version

contract Example {                // Contract name
   function Example(...args) {} // Constructor

   function oneExample() {}       // method

   function() {}                  // fallback function
}
```

Hello World Example

```
pragma solidity ^0.4.15;

contract HelloWorld {
  event Print(string out, uint8 sum); // event that our Dapp will read
  function Example(string _toSay) {
    hello = _toSay;
  }
  string hello;// saved to storage
  uint8 sum = 5 + 2; // saved to storage
 // fallback function. Called whenever a contract receives a txn
 // with an invalid or no function signature given
 function() {
   Print(hello, sum); // event saved in receipt logs
 }
}
```

Data types

```
int a = 1;
uint256 b = 1 << 100;

bool isTrue = false;

string hello = "World";
bytes3 c = "lol";

enum MyChoices {Eat, Sleep, Repeat};

address myFriend = 0xc02c0307ab11559ca0cdcba1245439f95feafc14;

uint[3] = [1, 2, 12];
mapping(address => uint256) balances;

struct Person {
  string Name;
  uint8 age;
}
```

Functions and their accessors

```
function example() {};
function example() public {}
function example() public constant {}
function example() public view {}
function example() internal {}
function example() external {}
function example() private {}
function example() pure {}
function {}
```

Global Variables

msg	block	tx
sender	blockhash	gasprice
data	coinbase	origin
gas	difficulty	
sig	gasLimit	
value	number	
	Timestamp / now	

Address functions

```
address myAddress = 0x3f5CE5FBFe3E9af3971dD833D26bA9b5C936f0bE;
address myLibrary = 0x63f18e6418dc61D3B344D9Fe52810dbbB0336C35;

uint256 etherBalance = myAddress.balance;

myAddress.call(bytes4(keccak256("foo(uint256)")), ...args);
myAddress.call.gas(10000).value(1 ether)("someFunction", "someArgument");

myAddress.send(1 ether);
myAddress.transfer(1 ether);

myLibrary.callcode(bytes4(keccak256("myFunc(uint256)")), ...args);
myLibrary.delegatecall(bytes4(keccak256("myFunc(uint256)")), ...args);
```

Global Functions

```
assert(smt == true);
require(otherthing == false);
revert();


ecrecover(hash,v,r,s);
sha3(keccak256(("DoubleHash")));

addmod(x,y,k)
mulmod(x,y,k)

suicide(address myAddress);
selfdestruct(address myAddress);
```

Modifiers

- Used as a guard
- Runs **before** the function call
- Check arguments and/or valid state
- They are Inheritable properties
- Can be overwritten
- **Payable modifier**

```
modifier hasEnoughFunds() {
 require(msg.value > 1 ether);
 _;
}

modifier argumentIsValid(uint number) {
 require(number > 10);
 _;
}

function payMe() payable {
  lock = true;
  _;
  lock = false;
 }
```

Events

- Best way to interface between contract and dapp
- Cheap storage
- Arguments will be stored in a transaction log (receipts)
- Not accessible from inside contract
- Up to 3 Indexed params
- Inheritable

```
event SomethingHappened(byte32 arg1, uint arg2);
SomethingHappened("It really did", 42);

event SomethingHappenedAndICanSearchForIt(address indexed canSearch, uint noSearch);
```

Libraries

- Keeps code DRY
- Saves on gas
- Cannot be inherited
- Cannot receive Ether
- Using For

```
library myNewLib {

 function life(bytes8 otherArg) returns(uint8) {
   return 42;
 }
 function isTheMeaning(bytes8 arg) returns (bytes8){
   return "of";
 }
}

contract myContract {
 using myNewLib for bytes8;
 bytes8 what = "what";
 uint8 answer = what.isTheMeaning().life();
}
```

Inheritance

- Solidity Supports multiple inheritance
- Overloading is allowed
- Only 1 contract on blockchain
- “Super” calls parent class function
- Multiple inheritance

```
contract Life {
 uint searchFor;
 function Life(uint answer) {
  searchFor = answer;
 }

 function meaningOfLife() returns(uint) {
  return searchFor;
 }
}

contract Human is Life(42) {
 event Print(uint ans);
 bool isHuman = true;
 function meaningOfLife() returns(uint) {
  if (isHuman == true) {
   return super.meaningOfLife();
  }
  return 0;
 }
}
```

Creating a new contract Inside a contract

- Contracts can create other contracts
- But the code needs to be known before the contract can be created

Instantiating a contract inside a contract:

- Add the interface (or full code)
- Call the interface with the deployed address

```
contract Baby {
  function resetName(string newName);
}
contract Human {
Baby deployedBaby = Baby(0x0)
}
```

```
contract Baby {
  string name;
  function Baby(string givenName) {
      name = givenName;
  }
  function resetName(string newName) { name = newName; }
}

contract Human {
Baby[] public babies;
address[] public babyAddresses;
function createBaby(string name) {
   Baby myNewBaby = new Baby(name);
   babies.push(myNewBaby);
   babyAddresses.push(address(myNewBaby));
}
function lookAtBaby(uint index) public constant returns (Baby) {
   return babies[index];
}
 function lookAtBabyAddres(uint index) public constant returns(address) {
   return babyAddresses[index];
 }
}
```

Let’s code!

Challenges

- **Challenge 1:** Refactor the code so we only use 1 mapping
- **Challenge 2:** Bikers should be ablte to transfer credifts to a friend
- **Challenge 3:** As of right now, the Ether is locked in the contract and cannot move, make the Ether transferrable to your address immediately upon receipt

- **Advanced challenge 1:** Decouple the "database" aka mapping into another contract.
- **Advanced challenge 2:** Include an overflow protection library (or inherit from a contract)
- **Advanced challenge 3:** Develop an efficient way to track and store kms per rental, per user
- **Advanced challenge 4:** Add a repair bike bounty where the work can be claimed by a user and verified complete by another user (susceptible to attack?)
- **Advanced challenge 5:** Allow all users to vote on how many credits should be given for a donated bike within a time frame (susceptible to attack?)

Security and Design Patterns

- Re-entrancy
- Loops
- Sending/Receiving Ether
- .transfer vs .send vs, .call()
- Callstack depth
- Tx.origin
- Push over Pull

DAO

- What happened?
  - Hacker exploited a vulnerability to steal 160 million worth of Ether at the time
  - Completely drained the DAO
  - Led to **the** Ethereum hard fork splitting ETH/ETC
- How did it happen?
  - Used an easy exploit on the call() function
- Example