Lec 5: Solidity, a Smart Contract Language

Blockchain in the News

Akılalmaz olay... 'Bitcoin Safiye' her yerde aranıyor

Yazdır | Yazı Tipi | Yorumlar

Sadece Aydın'da değil, Türkiye genelinde yüzlerce kişiyi dolandırdığı ortaya çıkan kadın Safiye Gökçen Y.'nin insanları tuzağa düşürmek için kılıktan kılığa girdiği öğrenildi. Ülke genelinde aranan dolandırıcı için yurt dışına çıkış yasağı getirildi.

Sanal para birimi bitcoin borsasında vatandaşları dolandıran kadın broker Safiye Gökçen Y.'nin, sadece Aydın'da değil, Türkiye genelinde yüzlerce kişiyi dolandırdığı ortaya çıktı. Kadın borsacı kılıktan kılığa girerek yaklaşık 375 milyon TL ile sırra kadem bastı. Yeni Asır'ın haberinden sonra çok sayıda mağduru olduğu ortaya çıkan broker, yurt dışına çıkış yasağıyla Türkiye genelinde aranıyor.

En Çok Okunan Haberler

Son dakika: Cumhurbaşkanı Erdoğan'dan 17. yıl mesajı

Aidat kavgasında ortalık karıştı

Fujitsu pirinç ticaretine blokzincirini getiriyor

Blockchain 6 Kasım 2019

Fujitsu, İsviçre merkezli girişim Rice Exchange (Ricex) ile blokzinciri tabanlı pirinç ticareti platformu için iş birliğine gitti. Platformun 2020 yılında piyasaya sürülmesi bekleniyor. Küresel piyasa değeri 450 milyar dolar değerini aşan pirinç, dünyada ticareti en...

Devamını Oku

Coca-Cola blokzinciri çalışmalarını genişletecek

Blockchain 6 Kasım 2019

Coca-cola'nın tedarik zinciri operasyonlarını yürüten IT firması Coke One North America (CONA) blokzinciri teknolojisine yönelik çalışmalarını genişletmek istiyor. Business Insider'ın haberine göre SAP ile birlikte başarılı bir test süreci geçiren firma. Blokzinciri destekli bu programı...

Devamını Oku

Microsoft'tan blokzinciri tabanlı emlak platformuna destek

Blockchain 5 Kasım 2019

Hong Kong merkezli emlak ajansı Centaline, Microsoft ve TFI ile birlikte blokzinciri tabanlı emlak platformu geliştirdi. Centaline'nin mevcut uygulamasıyla entegre olan platform, değiştirilemez ve dijital olarak imzalanmış belgelere olanak tanıyor. 60.000 çalışanı ve 2000 şubesiyle...

Devamını Oku

Huawei, Çin'in dijital para araştırmalarına destek olacak

Blockchain 5 Kasım 2019

Çin'in blokzinciri teknolojisini hızlı bir şekilde benimsemeye çalıştığı şu günlerde teknoloji devi Huawei de Dijital Para Birimi Araştırma Enstitüsü ile anlaşma imzaladığını duyurdu. Huawei, Merkez Bankası'na bağlı Dijital Para Birimi Araştırma Enstitüsü ile anlaşmaya gittiklerini...

Devamını Oku

Çin, şehirler için kimlik tanıma sistemini başlatıyor

Blockchain 5 Kasım 2019

Çin, akıllı şehir altyapısının bir parçası şehirler için geliştirilen blokzinciri tabanlı bir kimlik tanımlama sistemi ile blokzincir alanındaki faaliyetlerini geliştirmeye devama diyor Global Times'ın haberine göre yeni geliştirilen kimlik tanıma sistemi Shijiazhuang şehrinde üç kurum...

Devamını Oku

Evolution of Blockchain Technology

- 1st generation: Store and transfer of value (e.g. Bitcoin, Ripple, Dash)
- 2nd generation: Programmable via smart contracts (E.g. Ethereum)
- 3rd generation: Enterprise blockchains (E.g. Hyperledger, R3 Corda & Ethereum Quorum)
- Next gen: Highly scalable with high concurrency (E.g. RChain)

Permissionless vs. Permissioned

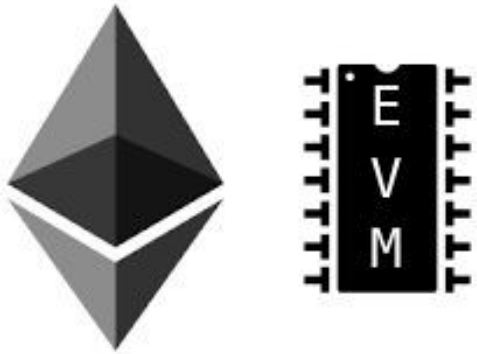
- Permissionless/Public
  - Decentralized Trustless Blockchains – Anyone can process blocks
  - Consensus typically reached via PBFT or Proof of Work
  - Miners compensated for validation
  - E.g. **Ethereum**, Bitcoin, Dash & Ripple
- Permissioned/Federated
  - Trusted: - All processors of blocks are known
  - Less processing required for consensus hence faster
  - E.g. **Hyperledger**, R3 Corda & Ethereum Quorum


Developing on Ethereum

THE ETHEREUM VIRTUAL MACHINE

THE WORLD COMPUTER

- Ethereum implements an execution environment known as the *Ethereum Virtual Machine*
  - Every node participating in the network runs the EVM
- Nodes will go through the transactions listed in the block they are verifying and run the code as triggered by the transaction within the EVM
- Every full node does the same calculations and stores the same values



OPCODES

WHAT MAKES THE COMPUTER TICK

See here for more

Value	Mnemonic	Gas Used	Subset	Removed from stack	Added to stack	Notes
0x00	STOP	0	zero	0	0	Halts execution.
0x01	ADD	3	verylow	2	1	Addition operation
0x02	MUL	5	low	2	1	Multiplication operation.
0x03	SUB	3	verylow	2	1	Subtraction operation.
0x04	DIV	5	low	2	1	Integer division operation.
0x05	SDIV	5	low	2	1	Signed integer division ope
0x06	MOD	5	low	2	1	Modulo remainder operatic
0x07	SMOD	5	low	2	1	Signed modulo remainder
0x08	ADDMOD	8	mid	3	1	Modulo addition operation.
0x09	MULMOD	8	mid	3	1	Modulo multiplication oper
0x0a	EXP	(exp == 0) ? 10 : (10 + 10 * (1 + log256(exp)))		2	1	Exponential operation.
0x0b	SIGNEXTEND	5	low	2	1	Extend length of two's com
0x10	LT	3	verylow	2	1	Less-than comparison.
0x11	GT	3	verylow	2	1	Greater-than comparison.

OPCODES

WHAT MAKES THE COMPUTER TICK

See here for more

Here given are the various exceptions to the state transition rules given in section 9 specified for each instruction, together with the additional instruction-specific definitions of J and C. For each instruction, also specified is α, the additional items placed on the stack and δ, the items removed from stack, as defined in section 9.

0s: Stop and Arithmetic Operations

All arithmetic is modulo 2^{256} unless otherwise noted.

Value	Mnemonic	δ	α	Description
0x00	STOP	0	0	Halts execution.
0x01	ADD	2	1	Addition operation. $\mu'_s[0] \equiv \mu_s[0] + \mu_s[1]$
0x02	MUL	2	1	Multiplication operation. $\mu'_s[0] \equiv \mu_s[0] \times \mu_s[1]$
0x03	SUB	2	1	Subtraction operation. $\mu'_s[0] \equiv \mu_s[0] - \mu_s[1]$
0x04	DIV	2	1	Integer division operation. $\mu'_s[0] \equiv \begin{cases} 0 & \text{if } \mu_s[1] = 0 \\ \lfloor \mu_s[0] \div \mu_s[1] \rfloor & \text{otherwise} \end{cases}$
0x05	SDIV	2	1	Signed integer division operation (truncated). $\mu'_s[0] \equiv \begin{cases} 0 & \text{if } \mu_s[1] = 0 \\ -2^{255} & \text{if } \mu_s[0] = -2^{255} \wedge \mu_s[1] = -1 \\ \mathbf{sgn}(\mu_s[0] \div \mu_s[1]) \lfloor \lvert \mu_s[0] \div \mu_s[1] \rvert \rfloor & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers. Note the overflow semantic when -2^{255} is negated.
0x06	MOD	2	1	Modulo remainder operation. $\mu'_s[0] \equiv \begin{cases} 0 & \text{if } \mu_s[1] = 0 \\ \mu_s[0] \bmod \mu_s[1] & \text{otherwise} \end{cases}$

THE ETHEREUM VIRTUAL MACHINE

THE WORLD COMPUTER

- Solidity lets you program on Ethereum, a blockchain-based virtual machine that allows the creation and execution of smart contracts, without requiring centralized or trusted parties
- Statically typed, contract programming language that has similarities to Javascript and C
  - Like objects in OOP, each contract contains state variables, functions, and common data types
  - Contract-specific features include modifier (guard) clauses, event notifiers for listeners, and custom global variables

Smart Contracts

- Execute on the Ethereum Virtual Machine (EVM)
- Languages:
  - Solidity: most popular, Turing complete, similar to JavaScript

```
contract Coin {
    address minter;
    mapping (address => uint) balances;
    function Coin() {
        minter = msg.sender;
    }
    function mint(address owner, uint amount) {
        if (msg.sender != minter) return;
        balances[owner] += amount;
    }
    function send(address receiver, uint amount) {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
    }
    function queryBalance(address addr) constant returns (uint balance) {
        return balances[addr];
    }
}
```

SOLIDITY -EXAMPLE

THE BANK CONTRACT

LEARNING FAST

WHAT DOES A BANK NEED TO DO?

1. Allow Deposits
2. Allow Withdrawals
3. Balance Checks

Learn X in Y minutes, a whirlwind tour of your favorite language

```
contract SimpleBank {
    mapping (address => uint) private balances;
    address public owner;
    event LogDepositMade(address accountAddress, uint amount);

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

    function deposit() public returns (uint) {
        balances[msg.sender] += msg.value;
        LogDepositMade(msg.sender, msg.value);
        return balances[msg.sender];
    }

    function withdraw(uint withdrawAmount) public returns (uint remainingBal) {
        if(balances[msg.sender] >= withdrawAmount) {
            balances[msg.sender] -= withdrawAmount;
            if (!msg.sender.send(withdrawAmount)) {
                balances[msg.sender] += withdrawAmount;
            }
        }
        return balances[msg.sender];
    }

    function balance() constant returns (uint) {
        return balances[msg.sender];
    }

    function () {
        throw;
    }
}
```

THE BANK CONTRACT

LEARNING FAST

```
1  contract SimpleBank {
2
3
4
5      mapping (address => uint) private balances;
6
7
8
9
10     address public owner;
11
12
```

- **`contract`** has similarities to **`class`** in other languages (class variables, inheritance, etc.)
  - Declare state variables outside function, persist through life of contract
- **`mapping`** is a dictionary that maps addresses to balances
  - always be careful about overflow attacks with numbers
  - **`private`** means that other contracts can't directly query balances
  - but data is still viewable to other parties on blockchain
- **`public`** makes externally readable (not writeable) by users or contracts

THE BANK CONTRACT

LEARNING FAST

- **`event`** - publicize actions to external listeners
- **`Constructor`** - can receive one or many variables here; only one allowed

```
4    event LogDepositMade(address accountAddress, uint amount);
5
6    function SimpleBank() {
7        owner = msg.sender;
8    }
```

- **`msg`** provides details about the message that's sent to the contract
  - **`msg.sender`** is contract caller (address of contract creator)

THE BANK CONTRACT

LEARNING FAST

- **`deposit()`**
  - Takes no parameters, but we are still sending Ether!
  - **`public`** makes externally readable (not writeable) by users or contracts
  - Returns user's balance as an unsigned integer (**`uint`**)

```
14    function deposit() public returns (uint) {
15
16
17        balances[msg.sender] += msg.value;
18
19
20        LogDepositMade(msg.sender, msg.value);
21
22
23        return balances[msg.sender];
24    }
```

- **`balances[msg.sender]`**, no **`this`** or **`self`** required with state variable
- **`LogDepositMade`** event fired

THE BANK CONTRACT

LEARNING FAST

```
function withdraw(uint withdrawAmount) public returns (uint remainingBal) {

    if(balances[msg.sender] >= withdrawAmount) {

        balances[msg.sender] -= withdrawAmount;

        if (!msg.sender.send(withdrawAmount)) {

            balances[msg.sender] += withdrawAmount;
        }
    }
    return balances[msg.sender];
}
```

- **`Withdraw()`**
  - **`withdrawAmount`** parameter
  - Returns user's balance
- Note the way we deduct the balance right away, before sending
  - We do this because of the risk of a recursive call that allows the caller to request an amount greater than their balance
- Increment back only on fail, as may be sending to contract that has overridden 'send' on the receipt end

THE BANK CONTRACT

LEARNING FAST

- **balance():**
  - constant prevents function from editing state variables
  - Returns user's balance

```
function balance() constant returns (uint) {

    return balances[msg.sender];
}
```

THE BANK CONTRACT

LEARNING FAST

() : Fallback function - Called if other functions don't match call or sent ether without data

- Typically, called when invalid data is sent
- Ether sent to this contract is reverted if the contract fails otherwise

```
function () {

    throw;
}
```

- **throw :** throw reverts state to before call

Solidity Language

Layout of a Solidity Source File

Pragmas

```
pragma solidity ^0.4.0;
```

Importing other Source Files

```
import "filename";
```

Structure of a Contract

State Variables

State variables are variables whose values are permanently stored in contract storage.

```
pragma solidity >=0.4.0 <0.6.0;
contract SimpleStorage {
        uint storedData; // State variable
        // ...
}
```

Functions

Functions are the executable units of code within a contract

```
pragma solidity >=0.4.0 <0.6.0;
contract SimpleAuction {

        function bid() public payable {
                // Function // ...
        }
}
```

DATA TYPES

INTEGERS

```
// uint used for currency amount (there are no doubles or floats) and for dates (in unix time)
uint x;

// int of 256 bits, cannot be changed after instantiation
int constant a = 8;
int256 constant a = 8; //same effect as line above, here the 256 is explicit
uint constant VERSION_ID = 0x123A1; //A hex constant
```

DATA TYPES

INTEGERS

```
Read the docs

// For int and uint, can explicitly set space in steps of 8 up to 256; e.g. int8, int16, int24
uint8 b;
int64 c;
uint248 e;

// Be careful that you don't overflow, and protect against attacks that do
// No random functions built in, use other contracts for randomness
```

DATA TYPES

BOOLEAN, ADDRESS

```
bool b = true;

// Addresses - holds 20 byte/160 bit Ethereum addresses
// No arithmetic allowed
address public owner;
```

DATA TYPES

ADDRESS, SENDING ETHER

```
address public owner;
// All addresses can be sent ether
owner.send(SOME_BALANCE); // returns false on failure


if (owner.send(*20 ether*)) {}
// REMEMBER: wrap in 'if', as contract addresses have
// functions executed on send and these can fail
// Also, make sure to deduct balances BEFORE attempting a send, as there is a risk of a
recursivecall that can  drain the contract
```

DATA TYPES

WEI vs ETHER

- *Wei*

  It's a denomination, like cents to Dollars / pennies to Pounds / kuruş to TL:

  1: wei
  10^3: (unspecified)
  10^6: (unspecified)
  10^9: (unspecified)
  10^12: szabo
  10^15: finney
  10^18: ether

DATA TYPES

ADDRESS, SENDING ETHER

- Balance of the address in *Wei*

```
<address>.balance
```

- Send given amount of Wei to address, throws on failure

```
<address>.transfer(uint256 amount)
```

DATA TYPES

BYTES

```
// Bytes available from 1 to 32
byte a; // byte is same as bytes1
bytes2 b;
bytes32 c;

// Dynamically sized bytes
bytes m; // A special array, same as byte[] array (but packed tightly)
// More expensive than bytes1 - bytes32, so use those when possible
```

DATA TYPES

BYTES

```
// same as bytes, but does not allow length or index access (for now)
string n = "hello";

// stored in UTF-8, note double quotes, not single

// string utility functions to be added in future

// prefer bytes32/bytes, as UTF-8 uses more storage
```

DATA TYPES

BYTES

```
// Type inference
// var does inferred typing based on first assignment,
// can't be used in functions parameters
var a = true;
// use carefully, inference may provide wrong type
// e.g., an int8, when a counter needs to be int16
```

DATA TYPES

FUNCTION ASSIGNMENT, DEFAULT VALUES, DELETE, UNWRAP TUPLES

```
// Variables can be used to assign function to variable
function a(uint x) returns (uint) {
   return x * 2;
}
var f = a;
f(22); // call


// By default, all values are set to 0 on instantiation
// Delete can be called on most types
// (does NOT destroy value, but sets value to 0, the initial value)
```

DATA STRUCTURES

DATA STRUCTURES

ARRAYS

```
// Arrays
bytes32[5] nicknames; // static array
bytes32[] names; // dynamic array
uint newLength = names.push("John"); // adding returns new length of the array
// Length
names.length; // get length
names.length = 1; // lengths can be set (for dynamic arrays in storage only)
// multidimensional array
uint x[][5]; // arr with 5 dynamic array elements (opposite order of most languages)
```

DATA STRUCTURES

MAPPINGS

```
// Dictionaries (any type to any other type)
mapping (string => uint) public balances;
balances["charles"] = 1;
console.log(balances["ada"]); // is 0, all non-set key values return zeroes
// 'public' allows following from another contract
contractName.balances("charles"); // returns 1
// 'public' created a getter (but not setter) like the following:
function balances(string _account) returns (uint balance) {
    return balances[_account];
}
```

DATA STRUCTURES

MAPPINGS

```
// Nested mappings
mapping (address => mapping (address => uint)) public custodians;


// To delete
delete balances["John"];
delete balances; // sets all elements to 0
// Unlike other languages, CANNOT iterate through all elements in
// mapping, without knowing source keys - can build data structure
// on top to do this
```

DATA STRUCTURES

STRUCTS

```
// Structs and enums
struct Bank {
    address owner;
    uint balance;
}
Bank b = Bank({
    owner: msg.sender,
    balance: 5
});
```

DATA STRUCTURES

STRUCTS

```
// or
struct Bank {
    address owner;
    uint balance;
}
Bank c = Bank(msg.sender, 5);
delete b;
// sets to initial value, set all variables in struct to 0, except mappings
```

DATA STRUCTURES

ENUMS

```
// Enums
enum State { Created, Locked, Inactive } // often used for state machine
State public state; // Declare variable from enum
state = State.Created;

// enums can be explicitly converted to ints
uint createdState = uint(State.Created); //  0
```

WHERE DATA GOES

NOTE ON MEMORY AND STORAGE

- Data locations: `memory` vs. `storage` vs. *stack* - all complex types (arrays, structs) have a data location
  - `memory` does not persist, `storage` does
- Default is `storage` for local and state variables; `memory` for function arguments
  - For most types, the data location to use can be explicitly set
- The stack holds small local variables
  - Used for values in intermediate calculations
  - General consensus is not to interact with it as a developer

GLOBAL VARIABLES

this

```
this; // address of contract

this.balance; // often used at end of contract life to send remaining balance to party
this.someFunction(); // calls func externally via call, not via internal jump
```

GLOBAL VARIABLES

msg, tx

```
// msg - Current message received by the contract
msg.sender; // address of sender
msg.value; // amount of ether provided to this contract in wei
msg.data; // bytes, complete call data
msg.gas; // remaining gas

// tx - This transaction
tx.origin; // address of sender of the transaction, will always be a user
tx.gasprice; // gas price of the transaction
```

GLOBAL VARIABLES

block, storage

```
// block - Information about current block
now; // current time (approximately), alias for block.timestamp (uses Unix time)
block.number; // current block number
block.difficulty; // current block difficulty
block.blockhash(1); // hash of the given block - returns bytes32, only works for most recent
256 blocks
block.gasLimit();
```

FUNCTIONS

```
// Functions can return many arguments, and by specifying returned
// arguments name don't need to explicitly return
function increment(uint x, uint y) returns (uint x, uint y) {
    x += 1;
    y += 1;
}
// Call previous function
uint (a,b) = increment(1,1);
```

FUNCTIONS

```
// 'constant' indicates that function does not/cannot change persistent
vars
// Constant functions execute locally, not on blockchain
uint y;

function increment(uint x) constant returns (uint x) {
    x += 1;
    y += 1; // this line would fail
    // y is a state variable, and can't be changed in a constant function
}
```

FUNCTIONS

VISIBILITY

```
// These can be placed where 'constant' is, including:
// public - visible externally and internally (default)
// external - only visible externally
// private - only visible in the current contract
// internal - only visible in current contract, and those deriving from it

// Functions hoisted - and can assign a function to a variable
function a() {
    var z = b;
    b();
}

function b() {

}

// Prefer loops to recursion (max call stack depth is 1024)
```

FUNCTIONS

FALLBACK

```
// Fallback function - Called if other functions don't match call or
// sent ether without data
// Typically, called when invalid data is sent
// Added so ether sent to this contract is reverted if the contract fails
// otherwise, the sender's money is transferred to contract
function () {
    revert(); // reverts state to before call
}
```

EVENTS

EVENTS ARE AMAZING

```
// Declare
event LogSent(address indexed from, address indexed to, uint amount);

// note capital first letter

// Call
Sent(from, to, amount);
// For an external party (a contract or external entity), to watch:
Coin.Sent().watch({}, '', function(error, result) {
    if (!error) {
        console.log("Coin transfer: " + result.args.amount +
        "   coins were sent from " + result.args.from +
        "   to " + result.args.to + ".");
        console.log("Balances now:\n" +
            "Sender: " + Coin.balances.call(result.args.from) +
            "Receiver: " + Coin.balances.call(result.args.to));
    }
}
// Common paradigm for one contract to depend on another (e.g., a
// contract that depends on current exchange rate provided by another)
```

MODIFIERS

MODIFIERS ARE ALSO AMAZING

```
// C. Modifiers
// Modifiers validate inputs to functions such as minimal balance or
user auth;
// similar to guard clause in other languages

// '_' (underscore) often included as last line in body, and indicates
// function being called should be placed there
modifier onlyAfter(uint _time) { if (now <= _time) throw; _ }


modifier onlyOwner { if (msg.sender == owner) _ }
// commonly used with state machines


modifier onlyIfState (State currState) { if (currState != State.A) _ }
```

MODIFIERS

MODIFIERS ARE ALSO AMAZING

```
// Append right after function declaration
function changeOwner(newOwner)
onlyAfter(someTime)
onlyOwner()
onlyIfState(State.A)
{
    owner = newOwner;
}

// underscore can be included before end of body,
// but explicitly returning will skip, so use carefully
modifier checkValue(uint amount) {

    _
    if (msg.value > amount) {
        uint amountToRefund = amount - msg.value;
        if (!msg.sender.send(amountToRefund)) {
            throw;
        }
    }
}
```

LOOPS

LOOPS ARE DANGEROUS

```
// All basic logic blocks work - including if/else, for, while, break, continue
// return - but no switch

// Syntax same as javascript, but no type conversion from non-boolean
// to boolean (comparison operators must be used to get the boolean val)

// For loops that are determined by user behavior, be careful - as contracts
have a maximal amount of gas for a block of code - and will fail if that is
exceeded
// For example:
for(uint x = 0; x < refundAddressList.length; x++) {
    if (!refundAddressList[x].send(SOME_AMOUNT)) {
       throw;
    }
}
```

EXTERNAL CONTRACTS

USAGE

```
contract InfoFeed {function info() returns (uint ret) { return 42; }}

contract Consumer {
    InfoFeed feed; // points to contract on blockchain


    function setFeed(address addr) { // Set feed to existing contract instance
        feed = InfoFeed(addr);// automatically cast, be careful; constructor is not called
    }

    function createNewFeed() { // Set feed to new instance of contract
        feed = new InfoFeed(); // new instance created; constructor called
    }

    function callFeed() {
        // final parentheses call contract, can optionally add
        // custom ether value or gas
        feed.info.value(10).gas(800)();
    }
}
```

INHERITANCE

USAGE

```
// Order matters, last inherited contract (i.e., 'def') can override parts of
// previously inherited contracts
contract MyContract is abc, def("a custom argument to def") {

// Override function
    function z() {
        if (msg.sender == owner) {
            def.z(); // call overridden function from def
            super.z(); // call immediate parent overriden function
        }
    }
}

// abstract function
function someAbstractFunction(uint x);
// cannot be compiled, so used in base/abstract contracts
// that are then implemented
```

IMPORTS

USAGE

```
// C. Import

import "filename";
import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol";
import "github.com/Arachnid/solidity-stringutils/strings.sol";
// Importing under active development
// Cannot currently be done at command line
```

REVERT, REQUIRE, ASSERT

USAGE

The revert function can be used to flag an error and revert the current call

The require function should be used to ensure valid conditions, such as inputs, or contract state variables are met, or to validate return from calls to external contracts.

assert acts like it might in any other programming language. Use of revert require should guarantee that your assert does not fail

REVERT, REQUIRE, ASSERT

USAGE

assert(bool condition):

throws if the condition is not met - to be used for internal errors. (Generally used to stop very bad things from happening.)

require(bool condition):

throws if the condition is not met - to be used for errors in inputs or external components.

revert():

abort execution and revert state changes

SELFDESTRUCT

USAGE

```
// selfdestruct current contract, sending funds to address (often creator)
selfdestruct(SOME_ADDRESS);

// removes storage/code from current/future blocks
// helps thin clients, but previous data persists in blockchain

// Common pattern, lets owner end the contract and receive remaining funds
function remove() {
    if(msg.sender == creator) { // Only let the contract creator do this
        selfdestruct(creator); // Makes contract inactive, returns funds
    }
}

// May want to deactivate contract manually, rather than selfdestruct
// (ether sent to selfdestructed contract is lost)
```

STORAGE

TIPS

- Storage is expensive because it is permanent
- Things like multidimensional arrays are expensive
- Store things OFF THE BLOCKCHAIN **unless absolutely necessary**

STORAGE

TIPS

THERE ARE NO SECRETS ON THE BLOCKCHAIN

All data on the blockchain can be easily viewed by anyone in the world

UNITS OF TIME

USAGE

```
// Currency is defined using wei, smallest unit of Ether
uint minAmount = 1 wei;
uint a = 1 finney; // 1 ether == 1000 finney
// Time units
1 == 1 second
1 minutes == 60 seconds

// Can multiply a variable times unit, as units are not stored in a variable
uint x = 5;
(x * 1 days); // 5 days

// Careful about leap seconds/years with equality statements for time
// (instead, prefer greater than/less than)
```

Demo

http://remix.ethereum.org

Hello world

```
pragma solidity ^0.4.18;

contract Hello{

        string greeting;

        function Hello(string _greeting) public {
                greeting = _greeting;
        }

        function greet() public constant returns (string)
        {
                return greeting;
        }
}
```

READINGS

https://coursetro.com/courses/20/Developing-Ethereum-Smart-Contracts-for-Beginners

Finish the tutorial – 8 videos (not written lessons ,click on play button)

reference:

https://solidity.readthedocs.io/en/develop/solidity-in-depth.html

References

Slides mainly adopted from

- Blockchain @ Berkeley : https://blockchain.berkeley.edu/
- Blockchain @ Princeton : http://bitcoinbook.cs.princeton.edu/