ELEN E6883: An Introduction to Blockchain Technology

Lab Manual

Electrical Engineering Department
Columbia University

LAB 1: MetaMask, Ethereum and Blocks

1. Introduction

In Lecture 1, the professor explicitly introduced blockchain and cryptography. In order to give us a deep understanding on how blocks work in real-world application, this lab will show the logic behind building blocks by using MetaMask wallet and creating Ethereum transactions. If you want to find a more comprehensive guide for MetaMask, you can check MetaMask Docs at https://docs.metamask.io/guide/#why-metamask.

By creating Ethereum accounts and making several transactions in MetaMask, we will have an in-depth look at some properties of Ethereum transaction and Cryptographic Hashing in order to fully understand the authenticity and security of Ethereum transactions.

2. MetaMask

2.1 Introduction

MetaMask is a plug-in Ethereum crypto wallet for Chrome onboard users. Available as a browser extension and as a mobile app, MetaMask equips us with a key vault, secure login, and token wallet—everything we need to manage our digital assets. MetaMask provides the simplest yet most secure way to connect to blockchain-based applications. In this and following labs, we will use MetaMask to store and send tokens not only between our accounts, but also in smart contracts which will be covered in the following lectures and labs.

2.2 MetaMask Setup

Complete information and study guide about MetaMask can be found at its official website metamask.io. We need to choose the right browser (Chrome is recommended) and follow its installation instruction. When we are creating a new MetaMask account, here are some key points we need to pay attention to.

First of all, creating a new strong password is extremely important because it encrypts private key. As we discussed in the lecture, private keys give access to all of our Ether or other tokens. So, it is better to have a strong password here.

Secret Backup Phrase, which includes 12 mnemonic words, will pop out after setting up the password. We need to write this phrase on a piece of paper or store it in a secure location because secret backup phrase makes easier to back up and restore our account if we log out our account or accidentally clear browser history.

We are now able to use interact with MetaMask.

2.3 Deposit Ether

Following steps can be completed on either MetaMask website or its extension interface (we can enter the interface from browser's extension toolbar, which is on the top-left corner for Chrome). First of all, we have to choose a right network to make our first transaction. There are several options for the networks: main network, local host, custom RPC and test networks, which include Ropsten, Kovan, Rinkeby and Goerli. The default network is Main Ethereum Network where we can trade real Ether. In this and following labs, we are going to use Rinkeby Test Network (you may also use other test networks) in which we can use free test Ether from the third-party websites. Testing on Main Network may cost too much real Ether which is unaffordable for developer like us. By clicking 'Main Ethereum Network' in website or interface, it will give us several choices and we are going to choose 'Rinkeby Test Network' for our developers' environment.

TO DO 1:
Deposit some Ether in your MetaMask accounts.

By following the steps in faucet.rinkeby.io, anyone that has a Twitter or Facebook account may request Rinkeby Ether within the permitted limits. If you want to use tokens in other networks, you can search 'free ether on (network's name) test network' or click 'Deposit' in MetaMask interface.

2.4 Make A Transaction

In this section, we are going to make a transaction between our accounts. We only have one default account right now, but we definitely need to create more accounts.

By clicking the top-right account picture in MetaMask interface, we can see 'Create Account' button. By clicking that button and entering the account name, we will switch back to the account where we deposited Ether, click 'Send', enter an amount of Ether, select an account we want to transfer to and choose the speed to send Ether. Depending on the speed we choose, the transaction usually takes about 15 - 30 seconds.

While we are waiting for it to be completed, we can find this transaction in 'Queue' (or in 'History' if the transaction is finished). By clicking 'View on Etherscan', we will find transaction details including Transaction Hash, Status, Block, Timestamp, From, To, Transaction Value and Transaction Fee. As we can see, our transaction has been recorded in Rinkeby Network, and anybody in the Rinkeby can see our blocks.

TO DO 2:
Create several accounts and make some transactions between these accounts.

3. Cryptographic Hashing

3.1 Introduction

Cryptographic Hashing function plays an important role in blockchain and any digital currencies related with that. In this section, we will talk about how cryptographic hash works; however, discussion about its properties is necessary. By analyzing and understanding these properties, we will know how blocks are chained together and how Ethereum or other blockchain networks operate.

3.2 Properties of Cryptographic Hashing

Because we have discussed these properties in the Lecture 1, there will be no detailed explanation in here. These properties are:

1) Deterministic
2) Quick to compute
3) Impossible to go back
4) Small change in input results in big change in output
5) Different messages generate different hash values

TO DO 3:
Test some properties of cryptographic hashing in Section 3.2.

The first website below has two fields: the upper one is 'Data' and the lower one is 'Hash'. You are going to enter some value in 'Data' field and check 'Hash' value in order to test some of properties. If you are interested in mining block, blockchain, distributed blockchain, tokens or coinbase transactions, please watch the tutorial video in the second link and practice them in the links starting from the third one.

https://andersbrownworth.com/blockchain/hash
https://www.youtube.com/watch?time_continue=5&v=_160oMzblY8&feature=emb_logo
https://andersbrownworth.com/blockchain/block
https://andersbrownworth.com/blockchain/blockchain
https://andersbrownworth.com/blockchain/distributed
https://andersbrownworth.com/blockchain/tokens
https://andersbrownworth.com/blockchain/coinbase

4. Ethereum Transaction

4.1 Introduction

In this section, we are going to have more in-depth look at how Ethereum transactions work and how to make sure that these transactions are authentic. By analyzing and understanding a classical example of an Ethereum transaction, we will fully understand the composition and authenticity of Ethereum transaction.

4.2 Preparing for Transaction Programmatically

If we look at any library which can be used to programmatically send a transaction to the network, like Web3.js library which we are going to work later, we will see that a transaction object contains several parameters. Some of them are required and some of them are optional. Let's have a closer look at a classical transaction object which just like the transaction we sent in TO DO 2:

1) from: the account where we send Ether from;
2) to: (optional) the account where we send Ether to;
3) value: (optional) amount of Ether we send in Wei (1 Ether = 10^18 Wei);
4) gas: (optional) maximum amount of gas in a transaction;
5) gasPrice: (optional) amount that sender pays per computational step;
6) data: (optional) ABI byte strings.
7) nonce: (optional) integer of a nonce

First three items are relatively easy to understand. Gas is a special unit in Ethereum and it refers to the cost necessary to perform a transaction. Ethereum gas functions similarly as gasoline in the car. Gas limit determines how much gas we can use in a transaction, and if we used up all gas, the entire transaction will be terminated. Gas limit works similarly with fuel tank, the volume of fuel tank cannot be changed once a car is produced and so does gas limit which cannot be changed once the transaction starts. MetaMask wallet will automatically set gas limit as 21,000 units when we have some simple transfers, but when the transaction becomes more complex or requires more computational steps, gas limit should accordingly go up. Depending on different computational steps and functions that we call, gas prices for transaction vary from zero to a couple of thousands Gwei (1 Gwei = 10^9 Wei). If you want to find specific gas price, please check Appendix G. Fee Schedule at:

https://ethereum.github.io/yellowpaper/paper.pdf

Data is an ABI (Application Binary Interface) byte string using in Smart Contracts when they are deployed to the blockchain. And we will discuss about data in detail when we start to design Smart Contract. Nonce is an integer which is incremented every time a transaction is sent in order to avoid replay attacks.

4.3 Ethereum Transaction Signature

The professor mentioned Cryptographic Hashing Functions and Digital Signature with Elliptic Curve in Lecture 1. He also discussed User Identity (Private Key and Public Key) in Lecture 2. In this section, we are going to expand our views by analyzing the authenticity of Ethereum transaction. Following equations play important roles in producing authentic transaction:

1) A Transaction Object + Private Key => Signed Transaction
2) Private Key + Elliptic Curve Digital Signature Algorithm (ECDSA) => Public Key
3) Public Key + Keccak Hash=> Ethereum Account
4) Signed Transaction + ECRECOVER => Ethereum Account

As we discussed in Section 2, a transaction object contains different parameters. Elliptic Curve Digital Signature Algorithm (ECDSA) ensures that, besides the owner of Private Key, anyone cannot generate the owner's Private Key from his/her Ethereum Account. If you are interested in ECDSA, please find more information at:

https://cryptobook.nakov.com/digital-signatures/ecdsa-sign-verify-messages

Keccak Hash is best known as hash function. It is widely used for authentication, encryption and pseudo-random number generation. For an explanation on how Ethereum uses Keccak Hash function, specifically Keccak-256, to generate Ethereum Account address, please check tayvano's reply here:

https://ethereum.stackexchange.com/questions/3542/how-are-ethereum-addresses-generated

ECRECOVER stands for Elliptic Curve Recover Function and it transforms signed transaction to Ethereum Account. If you want to get more information about how it works, please check the following website:

https://gist.github.com/axic/5b33912c6f61ae6fd96d6c4a47afde6d

We can generate Ethereum Account in two different ways which can help us verify the authenticity of an Ethereum transaction.

TO DO 4:

Add transactions to a blockchain.

For safety and security consideration, we want to add transactions to a blockchain that only the owner of private key can create these transactions. Watch the tutorial video in the first link and practice in the links starting from the second one.

https://www.youtube.com/watch?time_continue=1&v=xIDL_akeras&feature=emb_logo
https://andersbrownworth.com/blockchain/public-private-keys/keys
https://andersbrownworth.com/blockchain/public-private-keys/signatures
https://andersbrownworth.com/blockchain/public-private-keys/transaction
https://andersbrownworth.com/blockchain/public-private-keys/blockchain

LAB 2: Remix, Solidity and Smart Contract

1. Introduction

In Lecture 4, the professor introduced basic concepts of Ethereum, its properties and applications. This lab will focus on one of the most important properties – Ethereum smart contract. In order to compile and successfully deploy smart contract, there are three things we need to fully understand: Solidity (high level contract language), Remix IDE (Integrated Development Environment) and smart contract itself.

First of all, we need to understand the differences between a paper contract and a smart contract and the reason why smart contracts become increasingly popular and important in recent years. A contract, by definition, is a written or spoken (mostly written) law-enforced agreement containing the rights and duties of the parties. Because most of business contracts are complicated and tricky, the parties need to hire professional agents or lawyers for protecting their own rights. However, if we hire those professionals every time we sign contracts, it is going to be extremely costly and inefficient. Smart contracts perfectly solve this by working on 'If-Then' principle and also as escrow services. All participants need to put their money, ownership right or other tradable assets into smart contracts before any successful transaction. As long as all participating parties meet the requirement, smart contracts will simultaneously distribute stored assets to recipients and the distribution process will be witnessed and verified by the nodes on Ethereum network.

There are a couple of languages we can use to program smart contract. Solidity, an object-oriented and high-level language, is by far the most famous and well maintained one. We can use Solidity to create various smart contracts which can be used in different scenarios, including voting, blind auctions and safe remote purchase. In this lab, we will discuss the semantics and syntax of Solidity with specific explanation, examples and practices. If you want to find more information about Solidity, please check its official website at: solidity.readthedocs.io.

After deciding the coding language, we need to pick an appropriate compiler. Among various compilers like Visual Code Studio, we will use Remix IDE in this and following labs because it can be directly accessed from browser where we can test, debug and deploy smart contracts without any installation. Remix can be reached at its official website: http://remix.ethereum.org/.

2. Smart Contract with Different Blockchain Networks

2.1 Introduction

In this section, we are going to deploy a simple smart contract and interact with different blockchain nodes. But before doing these, we need to activate some plugins in order to make Remix more manageable. We need to click 'Plugin Manager' at the left side of Remix interface and make sure that 'Deploy & Run Transactions' and 'Solidity Complier' are in 'Active Modules' (we don't need to manage other plugins at this moment, just keep them at default settings).

2.2 First Smart Contract

By clicking 'File Explorers' at the left side of Remix interface, we can see the list of smart contracts (or empty list) under 'Browser'. Clicking '+' next to 'Browser', we will start to compile our first contract by entering its name 'MyFirstContract.sol' (all solidity files need to add '.sol' in the end of the file name).

Before compiling smart contracts, we need to choose the right version of compiler. Full list of Solidity versions displays under 'Compiler' in 'Solidity Compiler' tab. We are going to switch to any version starting with '0.6.x' (we mainly use 0.6.10+commit.00c0fcaf in this and the following labs). Because Solidity made some breaking changes in every big version update, if we use version 0.7.x, 0.5.x, 0.4.x or even lower versions, Remix might show some errors on our codes.

The sample of our first smart contract 'MyFirstContract' is in 'LAB 2 Assignment.txt'. You need to copy and paste this contract into Solidity file which is created earlier. The lab also provides the tutorial video 'My First Contract.mov'. You need to check if the deployment process of your contract is similar with the video.

2.3 JavaScript VM, Injected Web3 and Web3 Provider

In 'Deploy & Run Transaction', we are able to choose three different environments: JavaScript VM, Injected Web3 and Web3 Provider. When we use Injected Web3 (before using it, please make sure that you have followed the instructions in LAB 1 and installed MetaMask), Remix will automatically connect to Ethereum wallet - MetaMask. It means that any cost, including transaction fee and gas, will reduce Ether in MetaMask account.

Although Injected Web3 is the most similar environment to the real-world transaction, this environment does not suit for developers like us because we need to wait every time we deploy a smart contract. To save time for testing and debugging, we want an environment where we can get the result right after clicking 'Deploy' button. JavaScript VM (Virtual Machine) is a relatively better execution environment that simulates blockchain in memory of the browser (be careful that reloading or closing website might default Remix to its initialized settings). When we deploy a smart contract in JavaScript VM, the environment will immediately give us a feedback. In this course, we will mainly use this environment because it not only saves time, but also provides several testing accounts which are convenient and costless.

Another fast way to deploy smart contract is via Web3 Provider, which represents the external application for blockchain node. In this and following labs, we will use Ganache which can be downloaded at https://www.trufflesuite.com/ganache (you may also use other external blockchain nodes). By clicking 'QUICKSTART ETHEREUM' at its initialized interface, we will see ten Ethereum testing accounts with 100 testing ether each. If we switch the environment to Web3 Provider in Remix, Remix will pop out a 'External node request'. We do not need to make change about the settings except for 'Web3 Provider Endpoint'. The last four number (8545) of endpoint should be replaced by the last four numbers (7545) of 'RPC SERVER' in Ganache (new 'Web3 Provider Endpoint' becomes http://localhost:7545). As Remix connects to Ganache, the list of accounts in Remix will automatically switches to the list of ten accounts in Ganache; consequently, any transaction in Remix will immediately be shown in Ganache with details. Ganache not only shows what is going on behind the transaction, but also creates a private network which is important if we want to create a Java Script or HTML application with an actual user interface to connect with.

3. Solidity Language Description

3.1 Introduction

In this section, we are going to discuss the structure of different smart contracts and the type of variables, units and expressions. Some of them are similar with other programming language, like Java and Python, some of them are totally different. By the end of this lab, you should be able to compile an integrated smart contract. In order to have a better understanding about the smart contract and its properties, this lab will not only provide detailed explanations about each property, but also offer some practices in each following subsection.

3.2 Variables: Integers, Booleans, Address, Balance and String

Solidity has detailed explanations about Integers, Booleans, Address, Balance and Strings. You can find them in following websites:

https://solidity.readthedocs.io/en/v0.6.10/types.html#integers
https://solidity.readthedocs.io/en/v0.6.10/types.html#booleans
https://solidity.readthedocs.io/en/v0.6.10/types.html#address
https://solidity.readthedocs.io/en/v0.6.10/types.html#members-of-addresses
https://solidity.readthedocs.io/en/v0.6.10/types.html#bytes-and-strings-as-arrays

Here are some points we need to pay attention to. First of all, all variables in Solidity are initialized by default. For example, (u)int = 0, bool = false and string = ''. Unlike other programming languages, strings in Solidity are special arrays. A string does not have a length or index-access. Because using strings costs huge amount of gas, Solidity has very few functions to do any string manipulation, which means that we should avoid using strings as much as we can.

Then, as we can see in contract 'MyFirstContract', 'public' state variable automatically has a getter function with the name of the variable. If we run the contract and click that 'public' variable, the system will show the value stored in this variable.

TO DO 1:
Create a new file 'WorkingWithVariables.sol' in Solidity, open 'LAB 2 Assignment.txt', copy and paste 'TO DO 1' into Solidity file, and finish TO DO with the instructions.

There are **4 tags** that you need to fill in **TO DO 1**. When you finish filling in, you need to deploy the smart contract and make sure that the deployment process is similar with the video 'TO DO 1 Deployment.mov'.

3.3 Address and Global Msg-Object

The following websites give specific interpretation about the members of address types and the properties of block and transaction. In address types, you need to focus on how to get the balance of an address and how to transfer ether from one account to another.

Remix has special variables and properties for provid information about the blockchain, functions and transaction. In this subsection, you need to understand how to use 'msg-object', especially 'msg.sender' and 'msg.value'.

https://solidity.readthedocs.io/en/v0.6.10/units-and-global-variables.html#members-of-address-types
https://solidity.readthedocs.io/en/v0.6.10/units-and-global-variables.html#block-and-transaction-properties

TO DO 2:
Create a new file 'SendMoney.sol' in solidity, open 'LAB 2 Assignment.txt', copy and paste 'TO DO 2' into that file, and finish TO DO with the instructions.

There is only **1 tag** that you need to fill in **TO DO 2**. When you finish filling in, you need to deploy the smart contract and make sure that the deployment process is similar with the video 'TO DO 2 Deployment.mov'.

3.4 Starting, Pausing, Stopping and Deleting

Solidity explains constructor function, require function and selfdestruct function in detailed examples. A constructor function is a special function that only executed once when we deploy smart contract, so it is a really good place to put all initialization logic. A require function can be used to check for conditions and throw an exception if the condition is not met. If the condition is not net, we can provide an error message string, which is not required. A selfdestruct function destroys the current contract, sends remaining funds to the given address and end execution environment. You can find these functions at:

https://solidity.readthedocs.io/en/v0.6.10/contracts.html#constructors

https://solidity.readthedocs.io/en/v0.6.10/control-structures.html#id4
https://solidity.readthedocs.io/en/v0.6.10/units-and-global-variables.html#contract-related

TO DO 3:
Create a new file 'StartStopPauseDelete.sol' in solidity, open 'LAB 2 Assignment.txt', copy and paste 'TO DO 3' into that file, and finish TO DO with the instructions.

There are **4 tags** that you need to fill in **TO DO 3**. When you finish filling in, you need to deploy the smart contract and make sure that the deployment process is similar with the video 'TO DO 3 Deployment.mov'.

3.5 Mapping and Struct

Mapping and struct are two special properties in Solidity. Mappings can be treated as hash tables that every key is mapped to a value. Struct is a collection of value types, mappings and/or arrays which we want to store in the smart contract. Solidity explains mapping types and structs with several examples which you can find at:

https://solidity.readthedocs.io/en/v0.6.10/types.html#mapping-types
https://solidity.readthedocs.io/en/v0.6.10/types.html#structs

TO DO 4:
Create a new file 'MappingStruct.sol' in solidity, open 'LAB 2 Assignment.txt', copy and paste 'TO DO 4' into that file, and finish TO DO with the instructions.

There are **2 tags** that you need to fill in **TO DO 4**. When you finish filling in, you need to deploy the smart contract and make sure that the deployment process is similar with the video 'TO DO 4 Deployment.mov'.

3.6 Error Handling

In this section, we are going to discuss three ways to trigger exception: 'require', 'assert' and 'revert'. The biggest difference between 'assert' and 'require' is that the former should be used when we check internal state of the contract and the latter should be used to ensure valid conditions that cannot be detected until execution time. 'Revert' function not only throws an exception, but also revert the current call. Solidity explains 'require', 'assert' and 'revert' functions in following websites:

https://solidity.readthedocs.io/en/v0.6.10/control-structures.html#id4
https://solidity.readthedocs.io/en/v0.6.10/control-structures.html#revert

TO DO 5:
Create a new file 'Exception.sol' in Solidity, open 'LAB 2 Assignment.txt', copy and paste 'TO DO 5' into that file, and finish TO DO with the instructions.

There is only **1 tag** that you need to fill in **TO DO 5**. When you finish filling in, you need to deploy the smart contract and make sure that the deployment process is similar with the video 'TO DO 5 Deployment.mov'.

3.7 View/Pure, Receive Function and Fallback Function

Both view and pure functions promise not to modify the state of the function, but we can read the state from view function instead of pure function. If you want to find more information about View/Pure, Receive Function, Fallback Function, and the difference between Receive and Fallback in Solidity v0.6.0 breaking changes, please check:

https://solidity.readthedocs.io/en/v0.6.10/contracts.html#view-functions
https://solidity.readthedocs.io/en/v0.6.10/contracts.html#pure-functions
https://solidity.readthedocs.io/en/v0.6.10/contracts.html#receive-ether-function
https://solidity.readthedocs.io/en/v0.6.10/contracts.html#fallback-function
https://solidity.readthedocs.io/en/v0.6.10/060-breaking-changes.html#semantic-and-syntactic-changes

TO DO 6:
Create a new file 'Functions.sol' in solidity, open 'LAB 2 Assignment.txt', copy and paste 'TO DO 6' into that file, and finish TO DO with the instructions.

There are **3 tags** that you need to fill in **TO DO 6**. When you finish filling in, you need to deploy the smart contract and make sure that the deployment process is similar with the video 'TO DO 6 Deployment.mov'.

3.8 Inheritance, Modifier and Importing

In this section, we are going to import derived smart contracts into main smart contract in order to make main contract less redundant. Despite making it more concise, we need to use modifiers to do necessary checks in order to make sure that functions, especially in main smart contract, meet the conditions. Solidity explains the syntax of inheritance, modifier and importing at following websites:

https://solidity.readthedocs.io/en/v0.6.10/contracts.html#inheritance
https://solidity.readthedocs.io/en/v0.6.10/contracts.html#function-modifiers
https://solidity.readthedocs.io/en/v0.6.10/style-guide.html#imports

TO DO 7:
Create two new files 'InheritanceModifierImporting.sol' and 'Owned.sol' in solidity, open 'LAB 2 Assignment.txt', copy and paste two contracts in 'TO DO 7' into their belonging files, and finish TO DO with the instructions.

There are **4 tags** that you need to fill in **TO DO 7**. When you finish filling in, you need to deploy

the smart contract and make sure that the deployment process is similar with the video ‘TO DO 7 Deployment.mov’.

3.9 Events and Return Variables

Solidity events give an abstraction on top of the EVM’s logging functionality. There are three main cases where events are used:
1) returning values from transaction;
2) triggering functionality externally;
3) cheap data storage.

Solidity gives specific explanation on how to use events and return variables at:

https://solidity.readthedocs.io/en/v0.6.10/contracts.html#events
https://solidity.readthedocs.io/en/v0.6.10/contracts.html#return-variables

TO DO 8:
Create a new file ‘Event.sol’ in solidity, open ‘LAB 2 Assignment.txt’, copy and paste ‘TO DO 8’ into that file, and finish TO DO with the instructions.

There are **2 tags** that you need to fill in **TO DO 8**. When you finish filling in, you need to deploy the smart contract and make sure that the deployment process is similar with the video ‘TO DO 8 Deployment.mov’.

LAB 3: Shared Wallet

1. Introduction

Combining the concepts of the smart contract and blockchain node in Lecture 4 with the Solidity language description in LAB 2, we are going to build an integrated Solidity project 'Shared Wallet'. More specifically, this project wants to build a shared system for both the owner and other participants. The owner has the right to deposit funds into this contract, to set the allowance for other participants and to withdraw any amount of available funds stored in smart contract. However, the participants only can withdraw tokens within the limited allowance.

The project not only is a coding challenge, but also can be used in real world cases. Shared wallet system represents a simple salary or budget system. The owner is an employer and the participants are employees who treat allowance as salary or business budget. The employer puts the funds into this system and has the right to set different salaries for every worker. The employer can withdraw the funds either to himself/herself or to his/her employees, while the workers can only withdraw salaries to themselves within the limit.

This lab has two Solidity files: 'Allowance.sol' and 'ShareWallet.sol'. The former one is the inheritance of the latter one. By the end of this lab, you should not only understand the logic behind building an integrated smart contract, but also proficiently use its structures in the future.

Before compiling and deploying the contract, you should choose proper developing environment which includes changing 'Compiler' to '0.6.10', turning on plugins and so on (if you forget how to do configuration, please find more information in LAB 1 and 2).

2. Allowance.sol

This Solidity file imports library 'SafeMath' and contract 'Ownable'; the former helps us to do arithmetic operations with added overflow checks and the latter provides functions and modifier which can be used in this Solidity file.

Although allowance.sol is not main contract, it needs to provide some important functions and preparation steps for main contract. Most of functions in this contract relate with allowance. Here is the list of functions we need to accomplish in this smart contract:

4. an isOwner function which can be used in this and main contracts;
5. a function for owner to set the allowance;
6. a modifier function;
7. a function to reduce the allowance.

TO DO 1:

Create a new file 'Allowance.sol' in solidity, open 'LAB 3 Assignment.txt', copy and paste 'TO DO 1 Contract Allowance' into that file, and finish TO DO with the instructions.

There are **3 tags** that you need to fill in **TO DO 1**.

3. SharedWallet.sol

This Solidity file also imports ‘SafeMath’ and contract ‘Ownable’ as inheritance. The contract has three main functions:

4) a function which helps the owner and participants to withdraw money from contract;
5) a function to remove function ‘renounceOwnership’ in contract ‘Ownable’;
6) a receive function to deposit funds.

TO DO 2:

Create a new file ‘SharedWallet.sol’ in solidity, open ‘LAB 3 Assignment.txt’, copy and paste ‘TO DO 2 Contract Shared Wallet’ into that file, and finish TO DO with the instructions.

There are **8 tags** that you need to fill in **TO DO 2**. When you finish filling in, you need to deploy the smart contract and make sure that the deployment process is similar with the video ‘LAB 3 Deployment.mov’.

LAB 4: Web3.js

1. Introduction

In Lab 2 and 3, we used Remix, an online IDE (Integrated Development Environment) for Solidity programming language, to learn the properties and syntax of smart contracts. Remix is a friendly open source tool for beginners not only due to its powerful and concise visual interface, but also because users can directly access smart contract editor in browser without any installation. Considering that developers need to set up local blockchain for development environment when they start Solidity programming, Remix also helps them, especially beginners, to save this intimidating process. For more experienced smart contract developers, Remix provides them with a convenient platform where they can test new features of Solidity or do experiment on part of complicated smart contracts.

However, it is hard to do testing in Remix even though it manages some plugins. Because Remix is beginner-friendly and easy to use, it is relatively not extensible. On the other hand, if we are working on complicated projects, it will be extremely difficult to finish those projects with just Solidity language. Therefore, we want to use web3.js (Web3 JavaScript) library, an Ethereum JavaScript API (Application Programming Interface), to programmatically access to deployed smart contracts. Although web3.js uses CLI (Command-Line Interface) which is not beginner-friendly, it provides JavaScript applications with a portal to directly interact with Ethereum nodes and network.

Instead of discussing about Ethereum mechanism and properties in Lecture 4 and 5, this lab will concentrate on web3.js, which is the port to the internet outside, and use it for interacting with Ethereum accounts and deployed smart contracts. Because web3.js uses CLI, most of interaction takes place in terminal and browser. If you want to find more detailed information about web3.js and its packages, please check it official website: https://web3js.readthedocs.io/en/v1.2.11/. However, this lab will not focus on developing frontend user interfaces or web pages.

2. Installation and Configuration of Node.js and Web3.js

In order to apply web3.js in our projects, we need to install node.js first. Node.js is a powerful and efficient JavaScript-based platform which can be download at: https://nodejs.org/en/download/ (please use its LTS version which is recommended for the most of users).

After successfully downloading node.js, we need to open a new terminal and type in following commands to check the version of node (node.js) and npm (node.js package manager):

```
$ node -v
$ npm -v
```

Then we need to create an empty folder 'lab-4' and enter this folder in terminal.

```
$ cd lab-4
```

By entering the following command, we will initialize our directory as a node project:

```
$ npm init -y
```

Now we need to check if the directory is initialized by standard 'package.json' file:

```
$ ls
```

We are going to officially download web3.js by running the following command:

```
$ npm install --save web3
```

The process will take some time because node.js package manager will help us download dependencies and JavaScript modules, and store everything in the folder 'node_modules'. When npm finishes download, we will see following information:

```
+ web3@1.2.xx
added xxx packages from xxx contributors and audited xxx packages in xxx s
```

Lastly, we need to check if three folders ('node_modules', 'package-lock.json' and 'package.json') are in folder 'node_modules' by entering:

```
$ ls
```

3. Using Web3.js to Transfer Ether from One Account to Another

3.1 Introduction

In this section, we are going to use web3.js to interact with RPC (Remote Procedure Call) sever (Ganache in this case) from out blockchain node. When we open Ganache and click 'Quick Start Ethereum', we can see 10 virtual Ethereum accounts with each having 100 ethers. Using web3.js on terminal, we will transfer an amount of ethers from one account to another. In order to connect and interact with Ganache, we need an API (Application Programming Interface). Ganache offers us an API which is the RPC server address (HTTP://127.0.0.1:7545); this address can easily be

found on the startup page of Ganache.

3.2 Transfer Ether

First of all, we are going to enter node by typing:

```
$ node
```

Next step is importing web3 by using require statement:

```
> let Web3 = require("web3")
```

Node will enter into 'node_modules' folder and search for a folder called 'web3'. Then node will store every dependency in the folder 'web3' into variable 'Web3'. Checking what is inside 'Web3', we are going to type the name of variable:

```
> Web3
```

Now we are going to use RPC sever address in order to connect web3 with Ganache.

```
> let web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:7547"))
```

If you want to find out the packages of web3, you can double click 'tab' on keyboard after entering following command (web3.js official website explains the functions of packages):

```
> web3.
```

If you are interested in what are inside each of packages, for example, you can double click 'tab' on keyboard after entering following command:

```
> web3.eth.
```

```
> web3.utils.
```

Before transferring ethers between accounts, we need to pick an account and designate it as the sender. Instead of directly copying an address from Ganache, we can browse the list of accounts by entering:

```
> web3.eth.getAccounts(console.log)
```

Console.log helps us to get the result even though promise is pending (if you want to find more

information about promise and console.log, you can check the answer in https://stackoverflow.com/questions/50836242/how-does-thenconsole-log-and-then-console-log-in-a-promise-chain). Now we are going to choose one of addresses as the sender and find out its balance with following command:

```
> web3.eth.getBalance("SENDER'S ADDRESS").then(console.log)
```

The previous result contains a lot of zeros because it is in unit 'wei', which is the smallest unit of ether. If we want to make the result more intuitional, we can convert it to unit 'ether' by using following command:

```
> web3.eth.getBalance("SENDER'S
ADDRESS").then(function(result){console.log(web3.utils.fromWei(result, "ether"))})
```

After checking the funds in sender's address, we are going to make a transaction from sender's address to recipient's address with an amount of ether. We will do this by entering:

```
> web3.eth.sendTransaction({from:"SENDER'S ADDRESS", to:"RECIPIENT'S
ADDRESS", value:web3.utils.toWei("AMOUNT", "ether")})
```

In Ganache, we can see an amount of ether has been transferred from sender to recipient. We also can check the information of this transaction in 'TRANSACTIONS' tab.

4. Using Web3.js to Interact with Smart Contracts

4.1 Introduction

In this section, we are going one step further and starting to interact with smart contracts. More specifically, instead of calling variables and functions in Remix, we will act as clients and use terminal to control smart contracts. However, before doing this, we need to set up the configuration and environment.

First of all, we need a simple smart contract to interact with. You can find the sample contract 'SampleContract' in 'SampleContract.txt', and all you have to do is copy and paste this contract into Solidity file (you may also use your own contract). This smart contract contains a public unsigned integer myUint and a setter function setMyUint. The compiler version should be set as '0.6.10+' and the environment should be changed to 'Web3 Provider'. Then we need to set 'Web3 Provider Endpoint' in 'External Node Request' to http://127.0.0.1:7545, which is same with the RPC server address in Ganache. The setup is finished, and we can deploy the contract.

4.2 Interacting with Smart Contract

First and foremost, we are going to enter node and connect web3 with Ganache.

```
$ node
> let Web3 = require("web3")
> let web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:7547"))
```

Before executing a message call transaction in terminal, we need to find out 'from', 'to' and 'data', which are corresponding to account address, contract address and encoded function signature. Both account address and contract address can be found in 'Deploy & Run Transactions' tab (we can copy account address by clicking button beside 'Account' and copy contract address by clicking button in 'Deployed Contracts'). When we call myUint in Remix, the console will show this call transaction sending off to the node. Clicking this transaction for detailed information, we can find the value of 'Input', which also is the first 8-digit of encoded function signature with '0x' at the front.

```
> web3.eth.call({from:"ACCOUNT ADDRESS", to:"CONTRACT ADDRESS",
data:"INPUT"}).then(console.log)
```

This gives us an encoded string of myUint, which is 66. However, instead of directly copying encoded function signature from Remix, we want to use keccak hash function to calculate encoded function signature in terminal.

```
> web3.utils.keccak256("myUint()")
```

The previous result is a 64-digit keccak hash of function signature with '0x' at the front; however, we only need '0x' and the first 8-digit of encoded function signature. Therefore, we need a sub string to meet the condition.

```
> web3.utils.keccak256("myUint()").substr(0,10)
```

Combining the previous command with message call transaction, we can get following command which should give us a same result as previous message call transaction:

```
> web3.eth.call({from:"ACCOUNT ADDRESS", to:"CONTRACT ADDRESS",
data:web3.utils.keccak256("myUint()").substr(0,10)}).then(console.log)
```

Now we are going one step further. We will use ABI (Application Binary Interface) array which save a lot of work. The value of ABI can be copied in 'Solidity Compiler' tab. We will create a new instance of this smart contract by entering:

```
> let contract = new web3.eth.Contract(ABI, "CONTRACT ADDRESS")
```

Then we are able to interact with this smart contract and find out the value of 'myUint' with following command:

```
> contract.methods.myUint().call().then(console.log)
```

Another method 'setMyUint' can help us to change the value of 'myUint'.

```
> contract.methods.setMyUint(NEW UINT).send({from:"ACCOUNT ADDRESS"})
```

Lastly, we need to repeat one of previous commands and check if 'myUint' is changed.

```
> contract.methods.myUint().call().then(console.log)
```

5. Using Web3.js with Chrome to Interact with Smart Contracts

5.1 Introduction

In this section, we are going to use Chrome (you may also use other type of browsers, but Chrome is recommended) to interact with our blockchain nodes. Before doing that, we need to download web3.js package for browser. Although we have downloaded 'node_modules' folder which contains web3.js, the browser cannot access to web3.js except for directly using the package that has everything packaged in. Therefore, inside the terminal of 'lab-4' folder, we can download this package by entering:

```
$ npm install web3.js-browser
```

This lab provides you with a html file 'index.html' which we can copy and paste it into 'lab-4' folder without any modification. Opening with Chrome, we can see an empty website which has already connected with web3.js-browser. Now we are ready to open developer console and type in commands.

Lastly, we need to check if Ganache and Remix are operating, because we still need to use them in interaction. If you closed them, you can re-open Ganache and deploy 'SampleContract' in Remix like we did in Section 4.

5.2 Interacting with Smart Contract

Most of interaction between Chrome and the smart contract is similar with what we did in Section 3 and Section 4. If you are still confused or hesitate about completing following TO DOs, please check the previous content.

TO DO 1:
Connect the browser with Ganache.

TO DO 2:
Get 10 Ganache accounts in developer console.

TO DO 3:
Create a variable 'contract' in order to connect browser with the smart contract in Remix:

Now we can check the value of myUint:

```
> contract.methods.myUint().call().then(result => console.log(result.toString()))
```

TO DO 4:
Change myUint to a new number (e.g. 2020) by calling setMyUint function:

TO DO 5:
Repeat one of previous commands and check if 'myUint' is changed.

LAB 5: Hyperledger Fabric

1. Introduction

In Lecture 9, we are given an introduction to Enterprise Blockchain, Hyperledger and Hyperledger Fabric. Now we understand the model of Hyperledger Fabric, its architecture and some of its key features. Considering that we have enough theoretical knowledge about Hyperledger Fabric, this lab offers a hands-on tutorial which includes running nodes in local machine and interacting with a Fabric test network.

More specifically, this tutorial will give us an explicit instruction on how to bring up the network, make transactions between nodes and finally bring down the network. First of all, we need to set up the environment by installing all the prerequisites which include Git, cURL, Go Language (optional but recommended) Docker and Docker Composer (if you are Window 10 user, you need to download extras).

Then we are going to open a terminal and bring up test network in a specific directory. Test network has three members: two peers and one orderer. As we learned in Lecture 9, peers are responsible for validating transactions and committing them to the blockchain ledger. Considering that, on a distributed network like Hyperledger Fabric, the differences in location and in view of transactions always create disagreements among peer nodes, it will be extremely costly if they want to reach the consensus. Orderer solves this problem by ordering signed transactions into blocks so that peers reduce expense and save time in reaching the consensus. There is only one orderer in this network; however, in real world, a network might have several orderers operated by multiple organizations.

After realizing the members of the test network and their functions, we are going to create a Fabric channel for transactions between organizations. The channel is private and only visible for the members invited to this channel. Each channel will have separate blockchain ledger which means that only the peers in the channel can store the channel ledger and validate the transactions.

Next step is starting chaincode on channel. As we learned in Lecture 9, Hyperledger Fabric chaincode refers to smart contracts which are deployed on the network in packages. After creating a channel, organizations in channel are able to interact with smart contracts for creating, governing and transferring assets on blockchain ledgers. However, to ensure the validity of a transaction, each organization in the channel needs to execute the smart contract on its peer node and signs the output of deployed smart contract. Once the output is consistent and the number of signatures meets the requirement, transaction will be committed to ledger.

With the previous preparation, we are able to set the environment variables and initialize the ledger with assets. Then we can query the peer's list of assets as well as change or transfer them to another peer by invoking a specific chaincode.

Finally, we need to bring down the network in order to stop and remove all crypto materials which includes all nodes (including 2 peers and 1 orderer), chaincode images and containers.

If you want to find more useful information about Hyperledger Fabric, please check at its official website: https://hyperledger-fabric.readthedocs.io/en/latest/index.html.

2. Fabric Test Network

2.1 Prerequisites installation and environment setup

First of all, we need to download the latest version of Git, cURL and Go Language (optional but highly recommended because Go is helpful when we start to use chaincode) which can be found in the following websites:

https://git-scm.com/downloads
https://curl.haxx.se/download.html
https://golang.org/

Next step is installing Docker which provides an operating platform for Hyperledger Fabric. For Mac OS, *nix, or Windows 10 users, you can download it at the first link. For older versions of Windows users, you can find the instruction to download Docker Toolbox at the second link.

https://www.docker.com/get-started
https://docs.docker.com/toolbox/toolbox_install_windows/

Installing Docker will also automatically install Docker Compose. We should check the version of Docker (version 17.06.2-ce or greater is required) and Docker Compose (version 1.14.0 or greater is required) with the following command from a terminal prompt:

```
$ docker --version

$ docker-compose --version
```

Be aware that if you are Window 10 users, you need some extra configuration steps. Before running any 'git clone' commands, you should run following commands:

```
$ git config --global core.autocrlf false

$ git config --global core.longpaths true
```

You can check the state of these parameters by entering:

```
$ git config --get core.autocrlf
```

```
$ git config --get core.longpaths
```

These need to be 'false' and 'true' respectively.

After downloading Docker and Docker Compose, we need to determine a location or create a folder where we want to put 'fabric-samples' repository in. Entering that folder from a terminal, we are going to run the following command:

```
$ curl -sSL https://bit.ly/2ysbOFE | bash -s
```

Previous command includes downloading the Hyperledger Fabric Docker images for the latest version and installing the Hyperledger Fabric platform-specific binaries and config files for the latest production release.

The setup is finished.

2.2 Bringing up the test network

Now we have installed all prerequisites, binaries and images. We can officially start to interact with Hyperledger Fabric test network. By typing in the following command, we can enter the 'test-network' directory in 'fabric-samples' folder.

```
$ cd fabric-samples/test-network
```

Before bringing up the network, printing the script help text gives us the first impression about this network, its modes and its flags.

```
$ ./network.sh -h
```

'./network.sh' represents the Hyperledger Fabric network which are using Docker on a local machine. Also inside this directory, we need to run the following command in order to check and remove all possible Docker containers or other artifacts from any previous runs.

```
$ ./network.sh down
```

Now it is time to officially bring up the test network.

```
$ ./network.sh up
```

If successful, we will see the similar information to this:

```
Creating network "net_test" with the default driver
```

```
Creating volume "net_orderer.example.com" with default driver
Creating volume "net_peer0.org1.example.com" with default driver
Creating volume "net_peer0.org2.example.com" with default driver
Creating peer0.org1.example.com ... done
Creating orderer.example.com    ... done
Creating peer0.org2.example.com ... done
CONTAINER ID   IMAGE                               COMMAND
CREATED                  STATUS                  PORTS
NAMES
26e05653a9dd   hyperledger/fabric-peer:latest      "peer node start"   Less
than a second ago   Up Less than a second   7051/tcp, 0.0.0.0:9051->9051/tcp
peer0.org2.example.com
e1bdf3a2efdc   hyperledger/fabric-orderer:latest   "orderer"           Less
than a second ago   Up Less than a second   0.0.0.0:7050->7050/tcp
orderer.example.com
1f79093369e9   hyperledger/fabric-peer:latest      "peer node start"   Less
than a second ago   Up Less than a second   0.0.0.0:7051->7051/tcp
peer0.org1.example.com
```

There are three nodes in the test network: two peers and one orderer. We have talked about their definitions and functions in both Lecture 9 and previous introduction section. The following command displays the list of components in this network:

```
$ docker ps -a
```

2.3 Creating a channel

Next step is creating a channel between two peers. This channel is a private layer for the communication and transaction between Org1 and Org2 who are invited into this channel. We can create the channel by running following command:

```
$ ./network.sh createChannel
```

The default name of this channel is 'mychannel'. If you want to create a channel with a specific name, you can try the following command:

```
$ ./network.sh createChannel -c channel1
```

2.4 Starting a chaincode on the channel

The lab explained the definition of chaincode in the previous introduction section. The following command will install the asset-transfer (basic) chaincode on Org1 and Org2 and then deploy the chaincode on default channel or a specified channel.

```
$ ./network.sh deployCC
```

2.5 Interacting with the network

To interact with our network, now we are able to use 'peer' CLI (command-line interface) which gives us an access to use deployed contract and update the channel. The following command will help us to add 'peer' binaries, which is in the 'bin' folder of 'fabric-samples' repository, to our CLI path.

```
$ export PATH=${PWD}/../bin:$PATH
```

At same time, we need to set 'FABRIC_CFG_PATH' to the 'core.yaml', which is in the 'config' folder of 'fabric-samples' repository.

```
$ export FABRIC_CFG_PATH=$PWD/../config/
```

We can now set the environment variables for Org1 with the following commands:

```
$ export CORE_PEER_TLS_ENABLED=true
$ export CORE_PEER_LOCALMSPID="Org1MSP"
$ export
CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.
example.com/peers/peer0.org1.example.com/tls/ca.crt
$ export
CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.exa
mple.com/users/Admin@org1.example.com/msp
```

```
$ export CORE_PEER_ADDRESS=localhost:7051
```

Then we need to initialize the ledger with the assets.

```
$ peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride
orderer.example.com --tls --cafile
${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.co
m/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n basic --peerAddresses
localhost:7051 --tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.
com/tls/ca.crt --peerAddresses localhost:9051 --tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.
com/tls/ca.crt -c '{"function":"InitLedger","Args":[]}'
```

If successful, we can see the result which are similar to this:

```
2020-08-14 18:17:41.809 EST [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001
Chaincode invoke successful. result: status:200
```

Now we can query the assets in our ledger with the following command:

```
$ peer chaincode query -C mychannel -n basic -c '{"Args":["GetAllAssets"]}'
```

If successful, we can see the output below:

```
[
   {"ID": "asset1", "color": "blue", "size": 5, "owner": "Tomoko", "appraisedValue": 300},
   {"ID": "asset2", "color": "red", "size": 5, "owner": "Brad", "appraisedValue": 400},
   {"ID": "asset3", "color": "green", "size": 10, "owner": "Jin Soo", "appraisedValue": 500},
   {"ID": "asset4", "color": "yellow", "size": 10, "owner": "Max", "appraisedValue": 600},
   {"ID": "asset5", "color": "black", "size": 15, "owner": "Adriana", "appraisedValue": 700},
   {"ID": "asset6", "color": "white", "size": 15, "owner": "Michel", "appraisedValue": 800}
```

]

With the assets in our ledger, we are able to make the transaction between two peers. Before doing that, we need to invoke the asset-transfer (basic) chaincode by running the following command:

```
$ peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride
orderer.example.com --tls --cafile
${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.co
m/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n basic --peerAddresses
localhost:7051 --tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example.
com/tls/ca.crt --peerAddresses localhost:9051 --tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example.
com/tls/ca.crt -c '{"function":"TransferAsset","Args":["asset6","Christopher"]}'
```

If successful, we can see a similar output to this:

```
2020-08-14 18:43:18.626 EST [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 001
Chaincode invoke successful. result: status:200
```

In order to query the asset of Org2, we need to set the environment variables by typing in the following commands:

```
$ export CORE_PEER_TLS_ENABLED=true
$ export CORE_PEER_LOCALMSPID="Org2MSP"
$ export
CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.
example.com/peers/peer0.org2.example.com/tls/ca.crt
$ export
CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.exa
mple.com/users/Admin@org2.example.com/msp
```

```
$ export CORE_PEER_ADDRESS=localhost:9051
```

Now we can query a specific asset stored in Org2 by running:

```
$ peer chaincode query -C mychannel -n basic -c '{"Args":["ReadAsset","asset6"]}'
```

If successful, we can see the output below:

```
{"ID":"asset6","color":"white","size":15,"owner":"Christopher","appraisedValue":800}
```

2.6 Bringing down the network

At this point, we finished the interaction with Fabric test network and we can now use the following command to bring down the network as well as stop and remove the nodes and containers.

```
$./network.sh down
```

TO DO:
Based on what we learned about smart contracts and Hyperledger Fabric in both lectures and labs, please write your first application by following the tutorial in https://hyperledger-fabric.readthedocs.io/en/latest/write_first_app.html.