## 3. *Understand Solidity Smart Contract in Blockchain*

**A.** IMPORTANT: Create a new genesis block by using **puppeth** command

1. Run **puppeth**
2. Give a *network-name*
3. Choose "Configure new genesis"
4. Choose "Create new genesis from scratch"
5. Choose "Ethash - proof-of-work"
6. Just press Enter to skip
7. Type "No" and press Enter
8. Give you chain ID
9. Choose "Manage existing genesis"
10. Choose "Export genesis configurations"
11. Give the path that you want to save your JSON file
12. Done! Use Ctrl + d to quit.
13. Use the *network-name.json* as the JSON file.
14. Running two peers and maintaining a new blockchain with the new genesis file.

**B.** We will use the Remix Solidity as the IDE for smart contract development.

1. The IDE home page is here: https://github.com/ethereum/remix-project
2. We suggest you just use the online one, which is easier: https://remix.ethereum.org.
3. If you want to work offline, you need to download and install it on your local computer.
4. In fact, you do not need to use this for a real project. We will use Truffle Suit with solidity plugins in IDEs like Intellij.

**C.** Basic Setup

1. Go to https://remix.ethereum.org
2. In another tab of your browser, go to https://remix-ide.readthedocs.io/en/latest/index.html
3. Please finish reading "New Layout Intro", "Tour of default modules", and "Tour of typical solidity modules" in the documentation.
4. Go back to the IDE tab, open the Plugin Manager and active the following modules: File explorers, Solidity compiler, Deploy and run transactions. You shall see their icons in the Icon Panel

**D.** Create and compile a new simple contract

1. We have learned a little about Solidity programming.
2. We will not go into Solidity language again in this tutorial. The official tutorial about Solidity can be found here: http://solidity.readthedocs.io/en/latest/index.html.
3. Attached with this tutorial, there is a file adder.sol. Open the file with a text editor and copy the content in the file in clipboard.
4. Add a new file in the contract folder with a file name "adder.sol".
5. Past the content of adder.sol in the new tab.

6. Use Ctrl + s to save the code. Go to the Solidity Compile to select Solidity 0.7.6 in the drop down list and recompile the code again.
7. You can turn on the Auto Compile if you want. The Enable optimization option is optional for now. It can optimize your code and produce high-performance bytecode if it is enabled.
8. In the bottom, look for the ABI and Bytecode. You can code the ABI and Bytecode by clicking them and pasting them in a text file editor. They will be automatically regenerated when the source code of your smart contract is updated and compiled. Please see and pay attention to the object field of the Bytecode. We will mention this again.
9. The compiled smart contract is ready to be deployed in a blockchain.

**E.** Deploy the smart contract in a Javascript VM

1. Click the "Deploy and run transactions" in the Icon Panel and the Side Panel is reloaded by another dashboard.
2. In the Side Panel, by default, the Javascript VM is selected for the default environment. It is a dummy blockchain as a simulator. You have 15 free accounts with 100 Ether in each of them. They can be used as the sender of your transactions to deploy the smart contract to the dummy blockchain. Feel free to choose any of them.
3. Every transaction requires a Gas Limit. The Gas Limit in the Side Panel is an upper bound of CPU cycles that you want to pay for miners. If the execution takes more than the upper bound, the translation will be discarded. Keep the default value should be fine.
4. Value in the  Side Panel should be 0. It is only used when you need to send Ethers to some account or smart contract.
5. The  Side Panel should detect that you have the adder smart contract ready to be deployed.
6. Like Java, the Java class code can be used to create many instances of the class. The smart contract file can be used as a template to create many instances of the smart contract. The deployed smart contract is similar to the object created in Java.
7. The smart contract file defines variables and functions, like java class file. Different instances of the same smart contract have different addresses and may maintain different states independently.
8. Functions can be used to access or modify the variable. The constructor is a special function that defines the initial state of a newly created instance of the smart contract.
9. Use keyword view on a function that does not change the state of the deployed smart contract, like the second function in the adder.sol example to return you the variable total maintained by the blockchain. The fourth function also has the view keyword because it does not change the state of the deployed smart contract either. You do not need to pay any transaction fee if you are calling a function that does not change the state of the blockchain.
10. One of the significant differences between smart contract and Java class is the ways how a new instance is created. Please try to click the Deploy button in the Side Panel, in the Terminal, you shall see the output of "mining".
11. Click the mined transaction in the Terminal. As I said, it is a dummy blockchain in your web browser, and the transaction deploying the smart contract is mined automatically.

You shall see the used gas amount and the decrease of Ether on the account selected for this transaction.

12. In the bottom of the Side Panel, notice that you have the "Deployed Contracts" appearing for ADDER AT *new-deployed-smart-contract-address*.
13. Click the arrow of the new address, you can now interact with the deployed smart contract. Every function that you have defined in the smart contract has a button created in the panel.
14. Try play the getTotal, you shall see 0 return back to you. Playing the getTotalConst button with an integer argument should give you the correct sum result back without changing the blockchain state value (i.e., the value of the total variable in our smart contract).
15. The yellow colour button means the operation that can change the blockchain state. Try clicking the addToTotal button with a number. A new transaction shall be mined in the simulator. Checking the getTotal again shall get you the updated value of the state variable.

**F.** Creating an instance of the smart contract adder.sol in your local blockchain

1. Review Four Questions:
   a) How to create an instance of a smart contract?
   An instance of a smart contract is created by sending a *transaction* into the network (e.g., public network, test network, private network). Roughly speaking, everything in blockchain is transaction and everything that you would like to do with blockchain must be done by sending a transaction.
   b) Will the instance of a smart contract be created immediately after the transaction is sent?
   No, the instance of the smart contract is created when the transaction has been mined by a miner. In other words, the miner will create a new block and the instance of the smart contract will be located inside the newly created block.
   c) How can I access the instance of the smart contract that has just been created?
   You will be provided the address of the deployed smart contract to find the location of the instance of the smart contract.
   d) Is there any difference between calling view/pure functions and calling state-update functions with the instance of the smart contract?
   Yes. Again, we need to keep in mind that everything in blockchain is a transaction, including the function callins that change the states. Since the state-update functions change the blockchain (i.e., variables) of the smart contract instance, the transaction of a function calling has to be mined (by a miner) before causing its effect.
2. Deploy your "adder" smart contract and create your first smart contract instance in real blockchain
   a) Prerequisites
      1) Have done all steps in Tutorial 1 and Tutorial 2 with the new genesis file that is created in the very beginning of the tutorial.
      2) Keep two peers running but not mining.
      3) The smart contract file must be compiled and has no error. Since the browser-solidity will automatically compile the code, if there are no

errors showing up, it means the smart contract file has been compiled successfully and is ready to be deployed.

4) For the first peer, which is running with `--http.port 9001`, must have enough Ethers in its local account, say HASH_ACNT_1.

**b)** Step by step deployment for a smart contract

1) In the Side Panel of Deploy & run transitions module, for the Environment dropdown list, choose Web3 Provider
2) There should be a dialog window pop up and please go to the bottom and locate the Web3 Provider Endpoint. For your case, it should be http://127.0.0.1:9001. Clicking the "OK" button.
3) If everything went well, you shall see the `chainId` appears (Costume (chainId) network). The Account shall show the account that you created in peer1 with the mined Ether amount.
4) Note that you have switched to a real blockchain, so the previous deployed smart contract in the dummy simulator is not accessible.
5) If you reach this step, it means the online IDE can talk with your running peer successfully.
6) Click the "Deploy" button. You probably will (actually you should) observe the following error message in the Terminal:

```
creation of adder pending...
creation of adder errored: Returned error: authentication needed:
password or unlock
```

7) Now, go back to the console of your first peer, i.e., the console where the peer with `--http.port 9001` is running, and use the following command to unlock your account:
`personal.unlockAccount(HASH_ACNT_1)`
Or, you can use the following command to list all the accounts on this peer:
`eth.accounts`
to find the index (starting with 0) of your HASH_ACNT_1 in all the listed accounts. Assume HASH_ACNT_1 is listed as the $x^{th}$ place in the list, you can also unlock this account by using
`personal.unlockAccount(eth.accounts[`$x-1$`])`
Then, enter your password to unlock this account.
8) Go back to your browser-solidity, and click the "Deploy" button again. Now, you should see the following message:

```
creation of adder pending...
```

9) Before you start the mining process, I suggest you switch back to the console where you unlocked your account and run the following command:
`eth.pendingTransactions`

You will see the transaction that you just created. It convinces us that creating an instance of smart contract is actually sending a transaction to the network (it is the private network, in our case).

**10)** More about the transaction listed by `eth.pendingTransactions`:

  **i.** The values of `blockHash` and `blockNumber` are `null`. This is because the transaction has not been mined, we will revisit this later after the transaction has been mined so you will see meaningful values here.

  **ii.** `from`: gives you the account ID that would like to pay for the deployment of the smart contract. And yes, you need to pay to create an instance of smart contract.

  **iii.** `gas`: also interpreted as the gas-limit, it is the maximum amount of gas allowed to use in deploying the smart contract (i.e., creating an instance of the smart contract). To better understand the concept of gas, let us rephrase the previous sentence like this: the maximum amount of CPU cycles allowed to use in deploying the smart contract.

  **iv.** `gasPrice`: The number of Wei that the account would like to pay for each gas. It is the linkage between gas (i.e., computational power consumption) and Ether (i.e, money).

  **v.** `hash`: The transaction ID, you can use its value to retrieve this transaction after the transaction got mined. Let us assume its value is TRANS_ID_HASH, we will use this value later after the transaction has been mined

  **vi.** `input`: This is the bytecode of the compiled smart contract. Its value shall be the same as the object field of the Bytecode copied from the bottom of the Solidity Compiler panel.

  **vii.** `nonce`: The number of transactions that the account (given by the `from` field) have been sent before. It can be interpreted as the index of the current transaction from the current account.

  **viii.** `to`: Since deploying a smart contract has not recipient, the value of `to` is `null`. When you send Ether from an account to another, the value of `to` would not be `null`.

  **ix.** `value`: We are deploying a smart contract but not sending Ether, so the value is 0.

**11)** Now, you can either use the current console, or go to the console of your second running peer, and execute the command

```
miner.start(1)
```

to start the mining process. You can also start the miners on both peers, and they will compete in mining blocks.

**12)** As long as the next block in the block chain is found, the transaction should get mined. I would suggest you execute

```
miner.stop()
```

to stop the mining process for now.

The new smart contract should be deployed in your real local blockchain. The "ADDER AT *new-deployed-smart-contract-address*" will be listed below the "Deployed Contracts"

**c)** Use the block explorer to track the mined translation. Locate and go into the block where the smart contract deploy transaction is mined. You search the *new-deployed-smart-contract-address* in the explorer. Do see the Code in this address.

**3.** Calling the functions/methods with the instance of the smart contract

**1)** We can see three functions that we have defined in the smart contract file. The one which has not shown is the constructor of the smart contract.

**2)** Remember we have introduced the variable TRANS_ID_HASH in previous. Now, go back to either one of the terminals where your peers are running, and use the following command to retrieve the information of your transaction that has been mined:

```
eth.getTransaction(TRANS_ID_HASH)
```

It will list the information of the mined transaction. You now should see the `blockHash` and `blockNumber` in the list.

**3)** You can call the three functions if you want. The first function `getTotal()` is a view function, meaning the variables in the instance of the contract would not be changed. Therefore, you do not need to pay gas to invoke this kind of function. Since in our adder.sol code, we initialize the variable `total` with number 0, we see the result of `getTotal()` is 0.

**4)** Try to input a number in the blank after the third function and click the function name `addToTotal`. We actually try to execute the function `addToTotal()` in the smart contract. You will probably get error message saying:

```
transact to adder.addToTotal pending ...
transact to adder.addToTotal errored: Error: Returned error: authentication
needed: password or unlock
```

This is because the function `addToTotal()` changes the variables in the smart contract and the computational resources would be consumed to mine a new block to update the blockchain, so your account should pay for it and authorization is required. Please unlock your account the same way when you were trying to deploy your smart contract. I will not repeat the steps here again.

**5)** After unlocking your account, click `addToTotal` again, it will show the following message

```
transact to adder.addToTotal pending ...
```

Start a miner and keep it running until the transaction is mined. Before the transaction is mined, you can see the transaction by using `eth.pendingTransactions`. After the transaction is mined, information of the transaction should be shown in block explorer.

**6)** Click `getTotal` again, you will see that the `total` variable in the deployed instance of smart contract has updated by your entered number.

**G.** If you have any problems when you try to reproduce the experiment, please contact me for help. Email: zwang@ece.ubc.ca

**H.** In the next tutorial, we will learn how to use command lines in console to execute functions in the smart contract.