

# Overview

- Ethereum & smart contracts
  - Solidity tutorial
  - Smart-contract programming
  - Ethereum virtual machine (EVM)
- Two Types of Accounts (address) on Ethereum
- **Externally owned accounts (EOA):** address =  $H(PK)$
  - **contracts:** address =  $H(CreatorAddr, CreatorNonce)$

## Smart contract


- smart contract is a user-defined program running on BKC.
- smart contract has its account (identified by an address), CA, described by the following.
  1. **Nonce:** it will increase every time this account sends a transaction;
  2. **Balance:** The amount of Ether it has now; it can receive or send Ether.
  3. **code:** the program of this smart contract
  4. **Data:** the storage variables in this smart contract
- Smart contract **address** is determined by **sender address** (who sends SC to blockchain) and **sender nonce** (which increases after each tx of this sender).

## EOA vs CA

- So Ethereum has two accounts: EOA and CA
- CA has its address but **no** private key; it has a state (nonce, balance, code, Storage Data)
- EOA has its address with a private key; it has state (nonce, balance)

## Ethereum transactions

- Two transactions:
  - from EOA to EOA to transfer Ether only
  - from EOA or CA, to CA, to deploy/execute
- CA; this tx inf is called a **message**.

- Two types of **Messages**:
  1. Given a smart contract C, tx e  deploy it to blockchain at address of CA.
  2. tx encodes data (CA address, function name, args) to execute the **deployed** contract on the blockchain.



## Smart contract life cycle

- **Develop code:**  
You write the source code of a smart contract
- **Compile code:**  
You compile the source code to bytecode.
- **Deploy code**  
You sends tx encoding your smart contract to blockchain
- **Execute code**  
Someone (including you) sends tx encoding the function (to be invoked) and its arg values.

## Solidity

## outline

- Ethereum & smart contracts
- **Solidity tutorial**
- Smart-contract programming
- Ethereum virtual machine (EVM)
- **Solidity:** A programming language for smart contract. The program runs on blockchain
- **Online Development Platform:** <https://remix.ethereum.org/>
- Reference Book:  
Andreas M. Antonopoulos and Dr. Gavin Wood, *Mastering Ethereum*, O'Reilly Media Inc, 2018.
- Online documentation (current version: **0.8.19**):  
<https://docs.soliditylang.org/>

**3 ->** H is **Keccak-256** hash function. For both addresses computation, it computes Keccak-256 hash and then outputs the last 20 bytes (i.e., 160-bit). The CreatorNonce is the number of transactions the CreatorAddr has done.

**9 ->** Another popular choice of development platform is **Truffle** (on your local computer) using the blockchain simulated by **Ganache**. Truffle provides a more comprehensive and feature-rich development environment suitable for larger projects, while Remix offers a lightweight and accessible web-based IDE for quick prototyping and experimentation.

Our teaching uses Remix to get away from the complication from things other than the smart contract itself.

**10 ->** **Pragma solidity** is compiler directive that tell you which version of solidity can be used to compile the program. In the above program, it indicates that at least solidity 0.8.2 but should not use solidity 0.9.0 or higher version. You might also see **solidity ^0.8.2** in a program, this asks to use the **exact** solidity 0.8.2 for the compilation. The file name better uses the contract name to be more informative.

## The first solidity program: Storage.sol

```

1 // SPDX-License-Identifier: GPL-3.0;
2 pragma solidity >=0.8.2 <0.9.0;
3
4 /**
5  * The contract has store() function to store the user-provided value in variable num;
6  * it has retrieve() function to display the stored value.
7  */
8 contract Storage
9 {
10     uint256 number;
11     function store(uint256 num) public {
12         number = num;
13     }
14     function retrieve() public view returns (uint256){
15         return number;
16     }
17 }

```

## Interpretation (cont)

Contract name

Member function

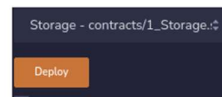
Member function

## Interpretation

- Integer:**
  - uint256 for unsigned integer of 256 bit; similarly uint8;
  - int256 for signed integer of 256 bit; similarly, int8.
- returns (uint256):** This indicates the **type** of the returned value of this function
- public:** This indicates the function `retrieve()` can be called externally.
- view:** Executing `retrieve()` should not change other variables in program

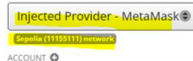
## Compile and Deploy a smart contract

- Compile (on Remix):**
- Deploy (Environment: **Remix VM**):** This will send tx to deploy the already compiled contract to the remix IDE's own blockchain. This is not a blockchain on the public network. It is a simulated blockchain only. This is fine at the develop stage.



## Deploy a contract on Ethereum Testnet

- Install **MetaMask** on your Chrome/Firefox.
- Login **MetaMask**
- Choose Environment: **Injected provider**: This will show one of your MetaMask account.



- Now press **deploy** button.

## Gas, Gas price and tx fee

- The tx sender needs to pay tx fee to the miner
- Recall:** Bitcoin tx fee is the difference of input and output values.
- On Ethereum blockchain, the  $tx\_fee = work\_amount \times price\_per\_unit\_work$
- work amount is called **gas**; it is the amount of computing work by miner in order to include a tx on blockchain
- price\_per\_unit\_work is called **gas price**; it is the amount of Ether the sender is willing to pay to the miner for each gas computation.

## Gas for some Ethereum Opcodes

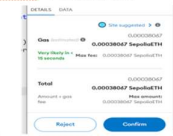
Stack Name	Gas	Initial Stack	Resulting St
00 STOP	0		
01 ADD	3	a, b	a + b
02 MUL	5	a, b	a * b
03 SUB	3	a, b	a - b
04 DIV	5	a, b	a // b
05 SDIV	5	a, b	a // b
06 MOD	5	a, b	a % b
07 SMOD	5	a, b	a % b
08 ADDMOD	8	a, b, N	(a + b) % N
09 MULMOD	8	a, b, N	(a * b) % N
0A EXP	10	a, b	a ** b
0B SIGNEXTEND	5	b, x	SIGNEXTEND(x, b)

You should regard this site as the source for what operations you can use in your contract.

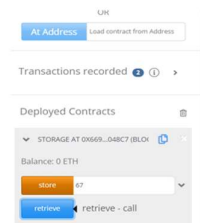
- <https://ethereum.org/en/developers/docs/evm/opcodes/>
- 0A exp to compute  $a^b$  with cost depends on b; details see <https://github.com/woolflo/evm-opcodes/blob/main/gas.md>

## Gas

- The gas for a tx is defined as  $gas = 21000 + X$ .
- 21000 accounts for the basic computing cost by miner:
  - verifying signature for tx and more
  - transfer Ether between different account cost 21000 gas.
- X counts for the cost executing the contract functions. This is determined by the basic operations in this contract function.



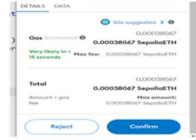
- This will ask your **MetaMask** for confirm



- Then, the tx to deploy your smart contract will be sent to a Ethereum Testnet.
- Here is my contract address: `0xf1542E15c31F02bD05728c8A46cBe1f08112816D` (to see tx, go to <https://sepolia.etherscan.io/>)
- You can also access this deployed contract: input the address in the box next to **At Address** and click and will see the contract as in the picture. Run store or retrieve function will send a tx to Ethereum blockchain

## How to Estimate Gas (fee)

- If you use MetaMask to send your tx, MetaMask will give you the gas amount estimate (where it in fact gives the gas fee).
- Here is the example for deploying `store.sol` on test `blockchain`.

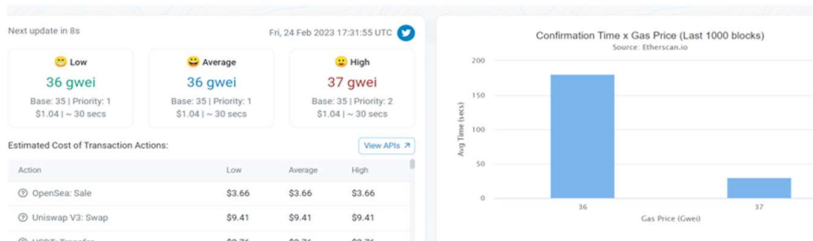


- If you can estimate gas with web3 (but we will not study this).

## Gas Price

- Gas price is determined by "market" (higher price allows your tx to be more likely to be included in a miner's block).
- You need to specify the price you want to pay.  
Price suggestion for main net, see

<https://etherscan.io/gastracker>



## A simple Faucet contract (to be explained)

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

    // Give out ether to anyone who asks
    function withdraw (uint withdraw_amount) public {
        // Limit withdrawal amount in WEI
        require (withdraw_amount<1000000000000000000);
        // Send the amount to the address that requested it
        payable(msg.sender).transfer(withdraw_amount);
    }

    // Accept any incoming amount
    function receiver() public payable {}
}
```

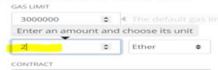
Condition to continue

Transfer Ether from contract address to address that invokes withdraw() function.

Contract can accept Ether.

## Transaction/Message: `msg` object

- `msg` is an object that basically represents the tx from the sender.
- If tx is to deploy a smart contract, then `msg` is an object contains the information of this deployment transaction.
- If tx is to invoke a function of contract, then `msg` is an object containing the information of function and its input, sender/receiver.
- `msg` has several attributes and methods (we study two here).
- `msg.sender`: the sender address object in tx (represented by `msg`)
- `msg.value`: the amount of ether sent to contract address.

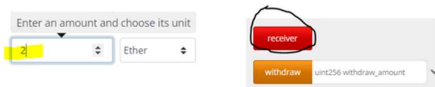


## payable keyword

- If a function has a keyword `payable`, then it can accept ether.

```
// Accept any incoming amount
function receiver() public payable {}
```

- After deploying, you can input 2 Ether and invoke `receiver` function to send it to the contract.



- `payable(alice)` makes `address alice` as `address payable` type.

## Gas limit and Execution Fail

- The gas estimate is hard to be accurate. It also depends on the execution.
- Sender specifies a **gas limit** he is willing to pay. If the execution is higher than the gas limit, the execution will stop but the used gas will not refund.
- If your gas usage is less than the limit, you only pay your usage.
- If the execution fails, the gas will be charged till the failure point.

## outline

- Ethereum & smart contracts
- Solidity tutorial
- **Smart-contract programming**
- Ethereum virtual machine (EVM)

## Address object: address

- There are two types of address: `address` and `address payable`
- Ex. `address payable alice`;
- It has some attributes or methods (we only give two).
- `alice.balance`: return the ether `alice` address has.
- `alice.transfer(178)`: this transfer 178 wei from the executed contract to `alice`.
- `address` (without `payable`) does not have transfer method.

## require statement

- Ex: `require(a<1000, "number a is too big")`
- It will check if `a<1000` is satisfied. If no, it will send error message "number a is too big".
- You can use `require(a<1000)` only (without the 2<sup>nd</sup> input).
- If `require` is not satisfied, the execution will throw an error and the state of contract will **revert** to the starting state. The gas will be charged till the failure point.

## contract constructor

- You can use constructor to initialize global variables

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

- Constructor is run only once.
- The sender that deploys the contractor will remain as the owner throughout the lifetime of this *deployed contract*.



**11 -> uint256** is the same as **uint** which can take the maximum value  $2^{256}-1$ . **int** (which is **int256**) takes value between  $-2^{255}$  to  $2^{255}-1$ . You might see **constant** function modifier. Somewhere. This is the older version of **view** type and deprecated now. **view** type function can be called without any cost (no gas usage and no signature) and you can call it by your local node (if it stores the whole blockchain) without broadcasting to the network.

**12 ->** Public function can be executed by any account (either an **EOA** or **CA**). In contrast, if you change **public** to **private** for retrieve ( ), then after deploy, it is the function is no longer accessible. You might also see **external** functions. External function can only be called externally while public function can be called by external and internal functions. You might also see **internal** function. Internal function can be called by current contract or its derived contract while private function can not be called by derived contract. Variable **number** (called storage variable, part of the contract state) can also be accessed by a **getter** function **number( )**.

**13 ->** When you are familiar with solidity, you can use Javascript tool Truffle to automatically deploy the smart contract.

**28 ->** You also have constructor with input: constructor (uint8 \_number) public { number=\_number; } . You will be asked to provide the input **\_number** when you deploy the contract.

**30 ->** The event usually is used to monitor the contract state change. For example, with event **tx\_sender** event, you know **owner's** balance has been changed.

**31 ->** nonce must match current nonce of sender in sender's account data. When Tx processed successfully, nonce in sender's account data is incremented by 1.

## function modifier

## Events

- **Question:** how to impose access control to a function execution?
- This can be done by **function modifier**.
- **ex.** want to modify faucet s.t. only the owner can withdraw.

```
modifier onlyOwner {require(msg.sender == owner, "only owner can call"); _; }  
  
function withdraw (uint withdraw_amount) public onlyOwner {  
  require (withdraw_amount<1000000000000);  
  payable(msg.sender).transfer(withdraw_amount);  
}
```

- It is possible to print some events for the contract execution.
- **ex.** when one deposits, we want to report the owner and the current balance.

```
event tx_sender(address, uint256);  
function receiver () public payable {  
  emit tx_sender(owner, address(this).balance);  
}
```

- use **event** to define an event
- use **emit** to output the event; you can see it from your **tx receipt**.

- Placeholder ( \_; ) is replaced by the code of the function that is being modified.
- **this** means the contract itself; **address(this)** is the contract address.

## Contract Transactions

- **To:** 32-byte address (to=0 means creating new account)
- **From:** 32-byte address
- **Value:** # Wei being sent with Tx
- **gasPrice, gasLimit:** Tx fees (later)
- **data:** what contract function to call & arguments  
if To = 0: create new contract
- **[signature]:** if Tx initiated by an external owned account

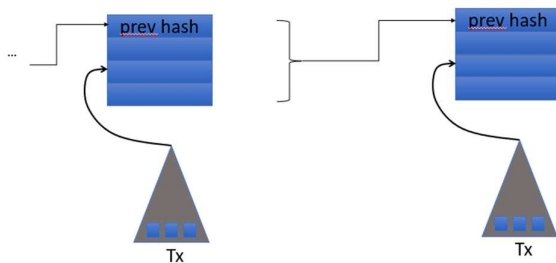
## Create New Block by Miner

Miners collect Tx from users:

⇒ run them sequentially on current world state

⇒ new block contains a list of **Txs**

## The Ethereum blockchain: abstractly



... EVM mechanics: execution environment

Write code in Solidity (or another front-end language)

⇒ compile to EVM bytecode

⇒ miners use the EVM to execute contract bytecode in response to a Tx

## The EVM

Stack machine (similar to Bitcoin)

- max stack depth = 1024
- miner keeps gas
- contract can create or call another contract

In addition: two types of storages

- **Persistent storage** (on **blockchain**): global variables
- **Volatile memory** (for single **Tx**): temporary variables