Gas refers to the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network. Since each Ethereum transaction requires computational resources to execute, each transaction requires a fee. Gas refers to the fee required to conduct a transaction on Ethereum successfully. Gas fees are paid in Ethereum's native currency, ether (ETH). Gas prices are denoted in gwei, which itself is a denomination of ETH - each gwei is equal to 0.000000001 ETH (10-9 ETH). Wei itself is the smallest unit of ETH. Total fee would have been: Gas units (limit) \* Gas price per unit. Before the London Upgrade, miners would receive the total gas fee from any transaction included in a block.

Gas is a reference to the computation required to process the transaction by a validator.

## AFTER THE LONDON UPGRADE

The total fee would now be: units of gas used \* (base fee + priority fee) where the base fee is a value set by the protocol and the priority fee is a value set by the user as a tip to the validator. Validator receives the tip of units of gas used \* priority fee. Base fee of units of gas used \* base fee is burned.

In short, gas fees help keep the Ethereum network secure. By requiring a fee for every computation executed on the network, we prevent bad actors from spamming the network. In order to avoid accidental or hostile infinite loops or other computational wastage in code, each transaction is required to set a limit to how many computational steps of code execution it can use. The fundamental unit of computation is "gas".

Although a transaction includes a limit, any gas not used in a transaction is returned to the user (i.e. max fee - (base fee + tip) is returned). Gas limit refers to the maximum amount of gas you are willing to consume on a transaction. A standard ETH transfer requires a gas limit of 21,000 units of gas.

For example, if you put a gas limit of 50,000 for a simple ETH transfer, the EVM would consume 21,000, and you would get back the remaining 29,000. However, if you specify too little gas, for example, a gas limit of 20,000 for a simple ETH transfer, the EVM will consume your 20,000 gas units attempting to fulfill the transaction, but it will not complete. The EVM then reverts any changes, but since the miner has already done 20k gas units worth of work, that gas is consumed.

High gas fees are due to the popularity of Ethereum. Gas price alone does not actually determine how much we have to pay for a particular transaction. To calculate the transaction fee, we have to multiply the gas used by the base gas fee, which is measured in gwei.

Transaction: Transactions are cryptographically signed instructions from accounts. An account will initiate a transaction to update the state of the Ethereum network. The simplest transaction is transferring ETH from one account to another.

An Ethereum transaction refers to an action initiated by an externally-owned account, in other words an account managed by a human, not a contract. For example, if Bob sends Alice 1 ETH, Bob's account must be debited and Alice's must be credited. This state-changing action takes place within a transaction. Transactions, which change the state of the EVM, need to be broadcast to the whole network. Any node can broadcast a request for a transaction to be executed on the EVM; after this happens, a validator will execute the transaction and propagate the resulting state change to the rest of the network.

* Contract deployment transactions: a transaction without a 'to' address, where the data field is used for the contract code.
* Regular transactions: a transaction from one account to another.
* Execution of a contract: a transaction that interacts with a deployed smart contract. In this case, 'to' address is the smart contract address.

Simple transfer transactions require 21000 units of Gas.

**Agenda**

1) Write a smart contract – A simple bank

Functionality:

1.a) Deposit some ether in the bank

1.b) Withdraw ether from the bank

1.c) Transfer ether from one account to another account

1.d) Check the balance corresponding to a account in the bank

2) Require an IDE such as remix online IDE, truffle, or any other to write smart contract and connect to the blockchain network

3) Deploy the smart contract onto a Ethereum blockchain network (mainnet, testnets, local private blockchain network such as Ganache)

3.a) Install Ganache

3.b) Install Metamask extension in the browser

3.c) Add the Ganache network in the metamask

3.d) Import Ganache accounts in the metamask

3.e) Connect the remix online IDE with the metamask

3.f) Select the Injected Provider – Metamask in the environment dropdown in remix online IDE

Note: If we use some testnet, we need to borrow some fake ether from the faucet (tool to get testnet ether).

4) Interact with the deployed smart contract

We will write a smart contract for depositing, withdrawing, and transferring functionality in a bank. This smart contract will act as a bank where we can deposit and withdraw the ether, and transfer the ether from one address to another address. We will also include a functionality to check the balance in an individual address.

As we know, there are two types of account addresses in Ethereum -

1) Externally-owned account (EOA) – controlled by anyone with the private keys

2) Contract account – a smart contract deployed to the network, controlled by code

Both account types have the ability to:

1) Receive, hold and send ETH and tokens

2) Interact with deployed smart contracts

* To write smart contracts, we require an IDE which allows us to write the smart contract and to connect to one of the Ethereum blockchain network. There are different IDEs available such as remix online IDE, truffle, and others.
* Once a smart contract is written, now we need to deploy this smart contract onto a Ethereum blockchain. A blockchain is nothing but a peer to peer network of nodes maintaining a synchronized ledger in the form of blocks connecting to each other through the hashes of the blocks. A block is nothing but a collection of transactions made by users or even smart contracts. These transactions are validated by the miners available in the blockchain network.
* So we need a Ethereum blockchain network through which users communicate. There are different networks available – mainnet (real Ethereum blockchain network where we need real ether to communicate in this network), testnet (different testnets are available such as Rinkeby, Goerli, and others where we need fake ether to communicate.), local private Ethereum blockchain network such as Ganache (where we need fake ether to communicate).
* Each node in the network is associated with an address (externally-owned account address) through which we communicate in the network. Ganache provides us default 10 accounts with 100 fake ethers. We can use these addresses to communicate with any other network as well. We need a wallet to maintain the ethers in the account. We need a private key to sign the transactions. We need a browser that can communicate with the blockchain network. Generally, all these are handled by a wallet called metamask itself. Metamask is a crypto wallet & a gateway to blockchain apps. MetaMask provides the simplest yet most secure way to connect to blockchain-based applications. MetaMask is an extension for accessing Ethereum enabled distributed applications, or "Dapps" in your browser! The extension injects the Ethereum web3 API into every website's javascript context, so that dapps can read from the blockchain. MetaMask generates passwords and keys on your device, so only you have access to your accounts and data.

Fundamentals and characteristics of Ethereum Blockchain: These Ethereum fundamentals form the foundation for understanding how to design and develop decentralized applications.

1. Ether and Networks: The Ethereum protocol has its own currency, called ether (denoted by ETH). The fundamental use of this currency is to pay block creators (minors or validators) to include, validate and execute the transactions in blocks. The smallest unit of a ether is called a wei (1 ether = 1018 wei). There are other units as well such as Gwei and others. While there is a single Ethereum protocol, there is more than one network running that protocol such as mainnet, testnet, and local private Ethereum network. The public Ethereum network is referred as “mainnet” where the ether has real-world value. There are public test networks or “testnets” available to test the smart contracts with fake ether. Testnets are just a replica of public Ethereum network. Each testnet has its own faucet used to provide fake ethers. It is also possible to create private networks running Ethereum, similar to how a private “internet” is called an “intranet”. For instance, Ganache is a tool that creates a local private Ethereum network with accounts having fake ethers. Ganache is kind of a simulator that allows to you to test the smart contracts and application. Ganache is mostly used for research purpose where you have to run 1000s of transactions at one go. Since ganache is a local network, it is fast compared to testnets.
2. Gas and Transaction Cost: There are different programming languages available to write the smart contract codes in Ethereum. Solidity is one of the popular programming languages. Like other programming languages, solidity also has a compiler which compiles the solidity code to bytecode, which provides a series of opcodes to the EVM. An opcode is an instruction such as PUSH 1 or MLOAD that is understood by EVM. Each of these opcodes has an associated gas cost. Note that a transaction contains bytecodes. The gas concept is used to decouple the price of ether from Ethereum’s transaction fees. To decouple the exchange rate between gas and ether, every Ethereum transaction sets its own gas price to determine how many wei a single unit of gas costs. As block creators decide which transactions to include in a block, they are incentivized to include the transactions that will give them the most generous gas price for their computations. The gas price is dynamic depending on the market rate. Therefore, you should set the gas price equal or above market rate so that the minors can select your transactions to be added in the block. Each Ethereum transaction has to include gas and gas price attributes, which when multiplied will set the maximum transaction fee for the transaction. This gas attribute sets a limit to how many computations the transaction can perform. If that limit is reached, the smart contract execution reverts, but the transaction is still written to the blockchain, and the fees are consumed by the block creator. If the smart contract call completes with leftover gas, the gas is returned to the transaction creator. Each block is specified with a gas limit. The sum of the gas used across all transactions in a block cannot exceed the block’s specified gaslimit. This also means that no single transaction’s gas usage can exceed the block’s gaslimit.
3. Accounts: There are two types of account in Ethereum blockchain. 1) Externally owned account (EOA): can send ether from one EOA to another EOA. 2) Smart contract account. Both are also identified by the Ethereum address. Additionally, Ethereum transactions can be sent from an EOA to smart contracts. Every transaction in the Ethereum blockchain is initiated by an EOA. Smart contracts can’t spontaneously perform an action. They can call other smart contracts, but every transaction originates from an EOA. When contracts are called, they can emit events, store data, receive ether, send ether to EOAs, or send data or ether to other contracts.
4. Contracts: Like object oriented parlance, a contract is really a class, or a collection of state variables, and functions. There are two types of solidity functions: read-only and write-only. Read-only functions are denoted with the pure and view keywords. Such functions cannot change the state of the contracts or emit events, therefore, can be called without paying any gas costs, and there will be no transaction created. There is no additional keyword with write-only functions. They can return data, but due to the asynchronous nature of Ethereum, the return data is practically useless. The function data must be sent via transaction and included in a block in order to for the function to be executed. They change the state of the contracts and often emits one or more events in the process. The purpose of events in Ethereum is generally twofold: to provide a custom historical log of what has occurred in the contract, and to allow observers to subscribe to real-time updates. Due to the open nature of blockchain, we already have a historical ledger of everything, but events provide more domain-specific logging and updates.
5. Transaction: A transaction can have an arbitrary number of contracts involved in its execution, provided that its execution fits within the constraints of the block’s gaslimit. Solidity exposes a number of other transaction-related attributes, but groups them into a message (msg) abstraction. Message refer to the communication between contracts and anything that can call them, such as other contracts. For example, a contract function call will always have a msg.sender. That msg.sender could be equal to the tx.origin or the creator of the transaction, or it could be the address of an intermediatory contract. Solidity’s message (msg) attributes are as follows: data (raw bytes of data sent to the currently executed function), sender (address of the caller of the currently executed external or public function), sig (a function identifier, first four bytes of the calldata determine which function is being called.), value (amount of wei sent to this function), timestamp, blockhash, difficulty, gaslimit, coinbase (address of block creator). To check the time reported by the computer the program is running on, we use block.timestamp. This is the time that the block was added to the blockchain. For every transaction in that block, the block.timestamp attribute will be identical. Due to a blockchain’s low-resolution clock, we can never expect second to occur. When you write code that checks the time, comparisons should always involve greater or less than, rather than exactly equal. It is also important to keep in mind that the block creator can manipulate the time a block is created as well as the ordering of the transactions to their advantage.
6. Signing transactions: The wallet software such as Metamask handles signing the transactions using the user’s private key. Knowing the EOA’s private key is synonymous with owning that account because the private key is what is used to sign transactions. Without this cryptographic signature, there is no way to authenticate whether a transaction was actually sent by its specified EOA. When we send an Ethereum transaction using any of the web3 libraries, the cryptographic signature happens in the background. The following transaction attributes are concatenated, encoded, and then signed with the configured private key (nonce, gasprice, gas, to, value, data, chainId). Once these attributes are signed, the signature itself is included in the transaction so that Ethereum nodes can validate that the sender is legitimate.

Ethereum Client:

In a traditional web application, the server is centralized and located with a URL or IP address. This web application software can be written in any programming language that is capable of sending HTTP request. The client software would make interacting with the server easier since it would contain all the logic and abstractions for building the requests and parsing responses. The client could also be released as a library and made available for other applications to speed up adoption of the service.

When we install an Ethereum client, we are installing software that will allow us to run an Ethereum node on our machine. This software comes with a command-line interface which allows us to create accounts or launch an interactive console that preloads Web3. Additionally, this server will run a server to expose the Ethereum JSON RPC API. We use JSON to send a request to a server, which then executes some predefined operations. It is through this JSON RPC that we will be interacting with the blockchain.

There are number of Ethereum clients you can use, including cpp-ethereum, Geth, Parity. Parity is a client written in rust and provides one of the faster syncing options of the available clients. Once the client is installed, you can begin to sync the blocks from the network. Goerli is a testnet designed to work with several different Ethereum clients such as geth and parity. After the initial sync, launching parity will be quick, as it will only need to grab the latest blocks that have been added to the chain. At this point, you can go ahead and kill the process with ctrl-C. If your application required users to download and run a full Ethereum client, you just lost a lot of potential users.

Metamask: If your application required users to download and run a full Ethereum client, you just lost a lot of potential users. Asking a user to install and run a full Ethereum node is a bit much for all. Outside of early adopter, you will need to provide a much easier way for those less-savvy to begin using the application, and that is exactly where MetaMask comes in. Metamask is distributed as a vrowser extension available on chrome and other browsers. The metamask software provides users with the ability to create accounts, and loads a preconfigured instance of web3 into browser that is used to interact with the blockchain via JSON RPC. With metamask installed, you can now begin to interact with existing decentralized applications from your browser.

Node.js: It is time to install node.js for the javascript tolls we will need for smart contract development. The primary development tools we will use to develop our smart contracts or interact with the Ethereum network have been built using JavaScript, which means we need Node.js to provide the JavaScript runtime environment.

Truffle (a framework): This provides tools make compiling, testing, deploying, and packaging your application as easy as possible. In order to deploy the contracts, we need to turn to another tool provided by truffle toolbelt, called migrations. Migrations are scripts written in JavaScript that are used to automate the deployment of our contracts. The default migrations contract found in contracts/Migrations.sol is the contract that is deployed by migrations/1\_initial\_migration.js and is currently the only contract that has made its way to test the network. We need to create another js file to deploy the created contract to the network. With truffle installed, the last thing we need to do is add Ganache.

Ganache: Its your very own blockchain. In many regards, it is very much like Ethereum client. It provides tools for creating accounts and runs a JSON RPC API server for you to connect and read/write to the blockchain. The main difference is that it doesn’t actually connect to the Ethereum network.

Solidity: In solidity, we don’t have access to standard out, or the file system, the network, or any other input/output, therefore can not print anything. What we do have are functions. The functions can return the values and on the front end, we can retrieve it and print it.

**Modifiers:** External functions can be called from other contracts, or from transactions, but cannot be called from within the contract or at least not without an explicit reference to the object it is being called on. The public functions are also part of the interface, meaning they can be called from other contracts or transactions, but additionally they can be called internally. This means you can use an implicit receiver of the message when invoking the method inside of a method. The internal and private functions must use the implicit receiver or, in other words, cannot be called on an object or on this. The major difference between these two modifiers is that private functions are only visible within the contract in which they are defined, and not in the derived contracts.

Functions that will not alter the state of the contract’s variables can be marked as either pure or view. The pure function do not read data from the blockchain. Instead, they operate on the data passed in or, in the case, data that did not need any input at all e.g. return a string “hello world”. The view functions are allowed to read data from the blockchain, but again they are restricted in that they can not write to the blockchain. We can indicate that the returned value is not referencing anything located in our contract’s persisted storage by using the keyword memory.

Function setGreeting (string calldata greeting) external {

}

Because this function is being called from the outside world, the data being passed in as a parameter is not part of the contract’s persisted storage, but is included as part of the calldata and must be labelled with the data location calldata. The calldata location is only needed when the function is declared as external and when the data type of the parameter is a reference type such as mapping, struct, string, or array.

State variables: they will be available to all the functions defined inside of a contract. They are also where we will store data that will exist for the entire lifetime of our contract. Like functions, state variables can be declared with different levels of visibility modifiers, including public, internal, and private. Note that all data on blockchain is publicly visible from the outside world. State variable modifiers only restrict how the data can be interacted with from within the contract or other contracts.

Making the greeter ownable:

We now add the idea of ownership to the contract, and then restrict the ability to change the greeting to the owner. In order to do this, we want to set the owner of the greeter contract to the address that deployed the contract. This means we will need to store the address during initialization, and for that, we will need to write a constructor function. We will also need to access some information from the msg object which is globally available. To check that owner exist, we can invoke an owner getter function. Since this is a getter function, we need to add a state variable that will hold the address of the owner, and then our function should return that address.

The solidity language provides two types of addresses: one is address and the other is address payable. The difference between them is that address payable gives access to the transfer and send methods, and variables of this type can also receive ether. We are not sending ether to this address and we can use the address type for our purposes.

Pragma solidity >= 0.4.0 < 0.7.0

Contract Greeter {

String private \_greeting = “Hello, World!”;

Address private \_owner;

Function greet() external view returns (string memory) {

Return \_greeting;

}

Function setGreeting(string calldata greeting) external {

\_grreting = greeting;

}

Function owner() public view returns(address){

Return \_owner;

}

}

Now, what we want to check is that the owner address is the same as the deploying address. Now we need to make a constructor that initialize the deploying address.

Constructor() public {

\_owner = msg.sender;

}

Now we know that who created the contract, we can create a restriction that only the owner can update the greeting. This type of access control is normally done with a function modifier. Such modifier will prevent the function from being invoked if the clause is not met. Our modifier will use the require function, where the first argument is an expression that will evaluate to a Boolean. When this expression results in a false, the transaction is completely reverted, meaning all stage changes are reversed and the program stops execution. The revert function also takes an optional string parameter that can be used to give more information to the caller as to why the operation failed. The last part of our modifier is the \_; line. This line is where the function that is being modified will be called. If you put anything after this line, it will be run after the function body completes.

Function setGreeting(string calldata greeting) external onlyOwner{

\_grreting = greeting;

}

Modifier onlyOwner(){

Require(msg.sender == \_owner, “Ownable: caller is not an owner”);

\_;

}

Import “openzeppelin-solidity/contracts/ownership/Ownable.sol”

Contract greeter is Ownable {

Rest code is same.

}

An import statement will pull in all the global symbols from the imported file, such as Ownable, and make them available in the current scope. Is keywork is used to inherit the properties from the imported file to the current contract. Solidity also supports multiple inheritance.

Contract deployment: We can deploy our smart contracts in the following three different ways:

1. Deploy smart contract to ganache, a local blockchain that will allow you to experiment with your application quickly.
2. Deploy the smart contract to the Goerli test network using the Ethereum client. This process is how you would deploy your application directly to the Ethereum network by using a node you are managing yourself.
3. Deploy the smart contract to the Rinkby test network using Infura, a third party provider of managed Ethereum nodes.

After compiling the smart contract, JSON file will be generated. There are several filed in this JSNO file. Two important fields are abi and bytecode fields. The application binary interface (ABI) describes the functions and events of the smart contract. The ABI will be the basis for the client-side abstraction used to interact with the smart contract. The bytecode field contains the result of the compiling the contract. This is the code the Ethereum network will execute when the contract has been invoked from a client.

What happens during the deployment: When we deploy the contract, we submit a transaction to the Ethereum network. The deployment transaction will need to set the receiving address to the 0x0 address. The deployment transaction will also include the bytecode, which will be sent as the transaction data. With the contract being sent as a transaction, it has to be mined before we will be able to interact with it. When the contract is mined, it will execute the code in the constructor, setting the initial state for the contract.

To deploy the smart contract, we need to configure the network (ganache) configuration in the truffle-config.js file. Once the ganache is running and configured the file, we will need to import the accounts into Metamask using the depicted mnemonic.

Tokens: Tokens are an abstraction that represents ownership. Ownership implies certain privileges such as the right to use or sell an item like a vehicle or a house. The token is represented by a title or deed. It can also provide the right to access something in the application. With ownership being something that can change often, tracking these changes on a cryptographically secured platform makes a lot of sense. To assist developers in the creation of tokens, the Ethereum community has developed several different types of token standards through the Ethereum Improvement Proposal (EIP) process. For example, ERC-20 and ERC-721

ERC-20: The ERC-20 standard is used when creating a fungible, or mutually interchangeable token. These tokens would be ideal replacements for things like reward points from retailers, miles from airlines, or a currency. All tokens created from an ERC-20 contract are considered to have the same value and are effectively indistinguishable from each other. Because all the tokens are considered identical, the primary responsibility of an ERC-20 contract is tracking balances. ERC-20 tokens have been used for many different purposes, but one that has likely caught your attention is the Initial Coin Offering (ICO). In an ICO, an organization will sell tokens as a means to raise funds.

If the token is expected to gain in value based on the performance of the issuing organization, the token may be considered a security. If that is the case, there are likely going to be some regulatory requirements that need to be considered when developing these smart contracts. See EIP-1462.