Lecture Notes

Smart Contract Development in Ethereum

Writing Smart Contracts in Solidity

In this session, you learnt about:

- Smart contracts and how to write them in Solidity
- Various components of Solidity
- Decentralised applications (DApps)
- Remix IDE, an online development environment

Introduction to Decentralised Applications

A decentralised application is a computer application that runs on a distributed computing system. **DApps** have been popularised by distributed ledger technologies such as the Ethereum blockchain, where they are often referred to as smart contracts.

DApps can be divided into four major layers as follows:

- **Frontend**
- **Backend**
- **Communications**
- **Database**

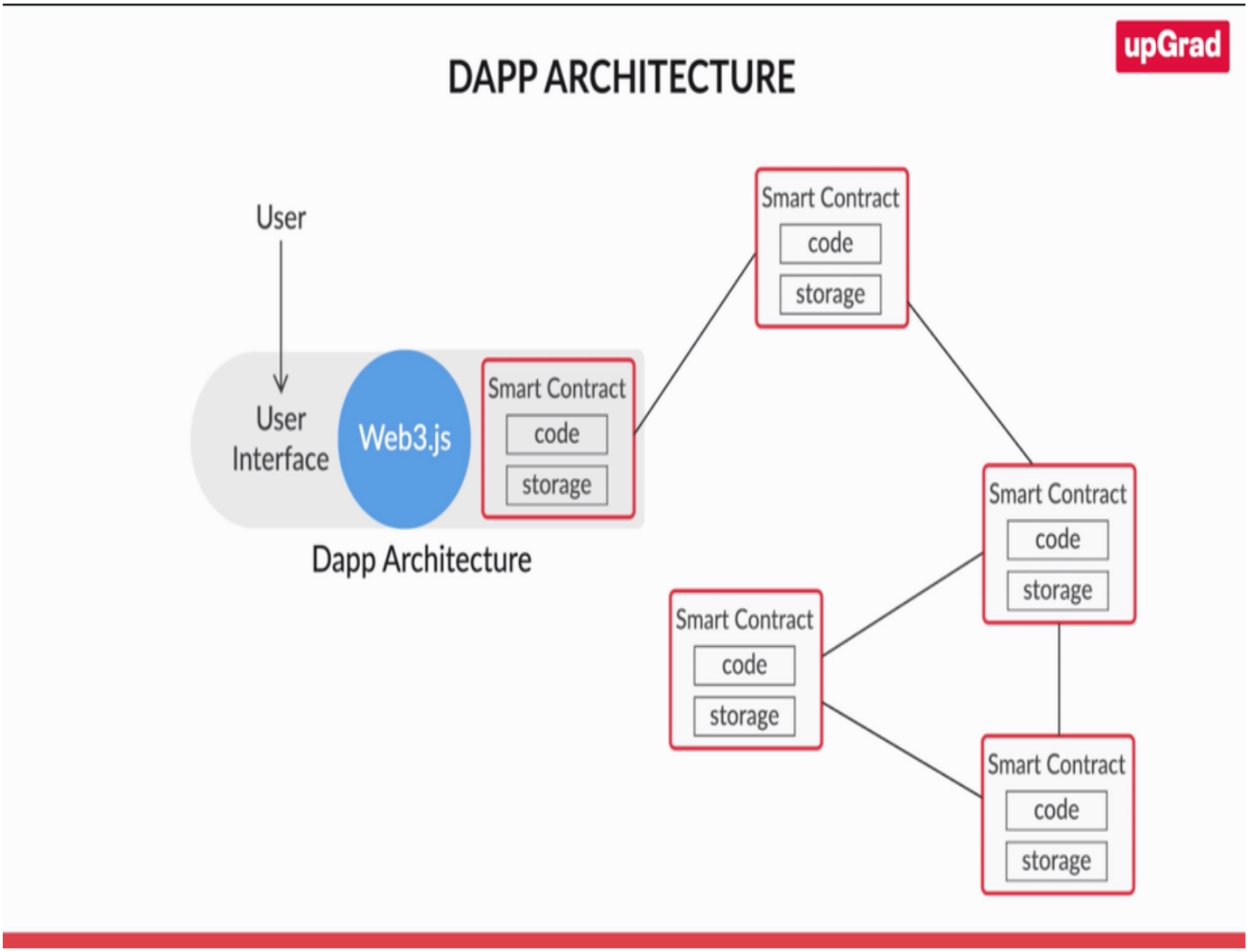
A distributed application also has the same structure. Ethereum too decentralises the backend or the logic layer of the distributed application. It does this by the means of smart contracts.

If every layer of a distributed application is decentralised, then that application becomes a completely decentralised application.

Following are the services needed to make a layer decentralised:

- **Frontend**: Swarm
- **Backend**: Ethereum
- **Communications**: Whisper
- **Database**

Architecture of a Decentralised Application



As shown in the image:

- Users interact with the application using the user interface, which is the frontend of the application.
- Web3.js is one of the tools that connects the frontend to the backend of the application.
- Smart contracts are installed over the blockchain nodes.

Introduction to Smart Contracts

A smart contract is a computer program or a transaction protocol which is intended to automatically execute, control, or document legally relevant events and actions according to the terms of a contract or an agreement.

Smart contracts are lines of code that are stored in a blockchain and automatically execute when predetermined terms and conditions are met. At the most basic level, they are programs that run as they have been set up to run by the people who developed them.

They **require** an open and decentralised database, which all parties of the **contract** can fully trust and which is fully automated. **Blockchains**, especially the Ethereum **blockchain**, are the perfect environments for **smart contracts**.

Smart contracts are immutable, that is, a contract once put on the network cannot be changed. It can only be deleted. On deleting a contract, only the contract code is deleted, not the contract account containing the contract code. If someone tries to access the deleted code and sends ether as part of the transaction, then those ethers will be lost. Hence, it is not advisable to delete smart contracts.

Smart contracts in Ethereum can only be triggered by the externally owned accounts (EOAs). Smart contracts cannot run in the background and they lie dormant until they are triggered by an EOA. Once triggered, they run as defined and then return to the dormant state.

Smart contracts are put on the Ethereum network using contract creation transactions. A new contract account is created and the contract bytecode is put in it. The address of that account becomes the 'id' for that smart contract, and anyone can access it using the id or the address.

Ethereum Virtual Machine (EVM)

Ethereum Virtual Machine (EVM) is a virtual machine that enables a computer or a machine to run smart contracts that are compiled to the EVM bytecode.
EVM is a runtime environment in which the smart contracts are executed. The smart contract codes need to be compiled to the EVM bytecode before they can be executed in the EVM. EVM is present on every node in the blockchain network.

Following are the two major properties of EVM:

- **Isolated:** The isolated nature of EVM enables users to run programs (smart contracts) without being affected by the system or platform on which they are run. It also ensures that the programs running are not affected by any other application on the system.

- **Sandboxed:** Being a sandbox, EVM is an isolated testing environment that enables users to run programs (smart contracts) without affecting the system or platform on which they run. Hence, they do not affect other applications on the platform.

The updating process of EVM is done through the EIP. EIP stands for Ethereum Improvement Proposal.
The Ethereum development community keeps making such proposals, which are mainly new features or bug fixes. If the majority of the Ethereum community accepts a proposal, then that EIP becomes a part of the next version of the EVM, which is then updated or installed on every Ethereum client or node.

The first version of EVM was called Frontier, and the current version of EVM is called St. Petersburg.

There are three locations for storing data in the EVM:

- Storage: Storage is a persistent storage space present in every Ethereum account.
- Memory: Memory is a volatile storage space.
- Stack: The EVM is a simple stack-based architecture. All the computations on the EVM are performed on the stack. The stack has a maximum capacity of 1,024 items and the size of each item on the EVM stack is 256 bits. If you run out of stack, then the contract execution will fail. The compiler generally uses it for intermediate values in computations and other scratch quantities. Once a contract execution is finished, the items containing the contract data are removed from the stack.

Introduction to Solidity

Solidity is an object-oriented, high-level programming language for implementing smart contracts. Solidity is highly influenced by C++, Python and JavaScript and has been designed to target the EVM.

Solidity codes are compiled to the EVM bytecode by the compiler. This EVM bytecode is then executed in the EVM.

Solidity is the most popular language for writing smart contracts among **Vyper**, **Serpent** and **LLL**.

Solc is the compiler that Solidity uses to compile the code to the EVM bytecode.

Remix is an online development environment that is used to write code (contracts) in Solidity.

Running Solidity:

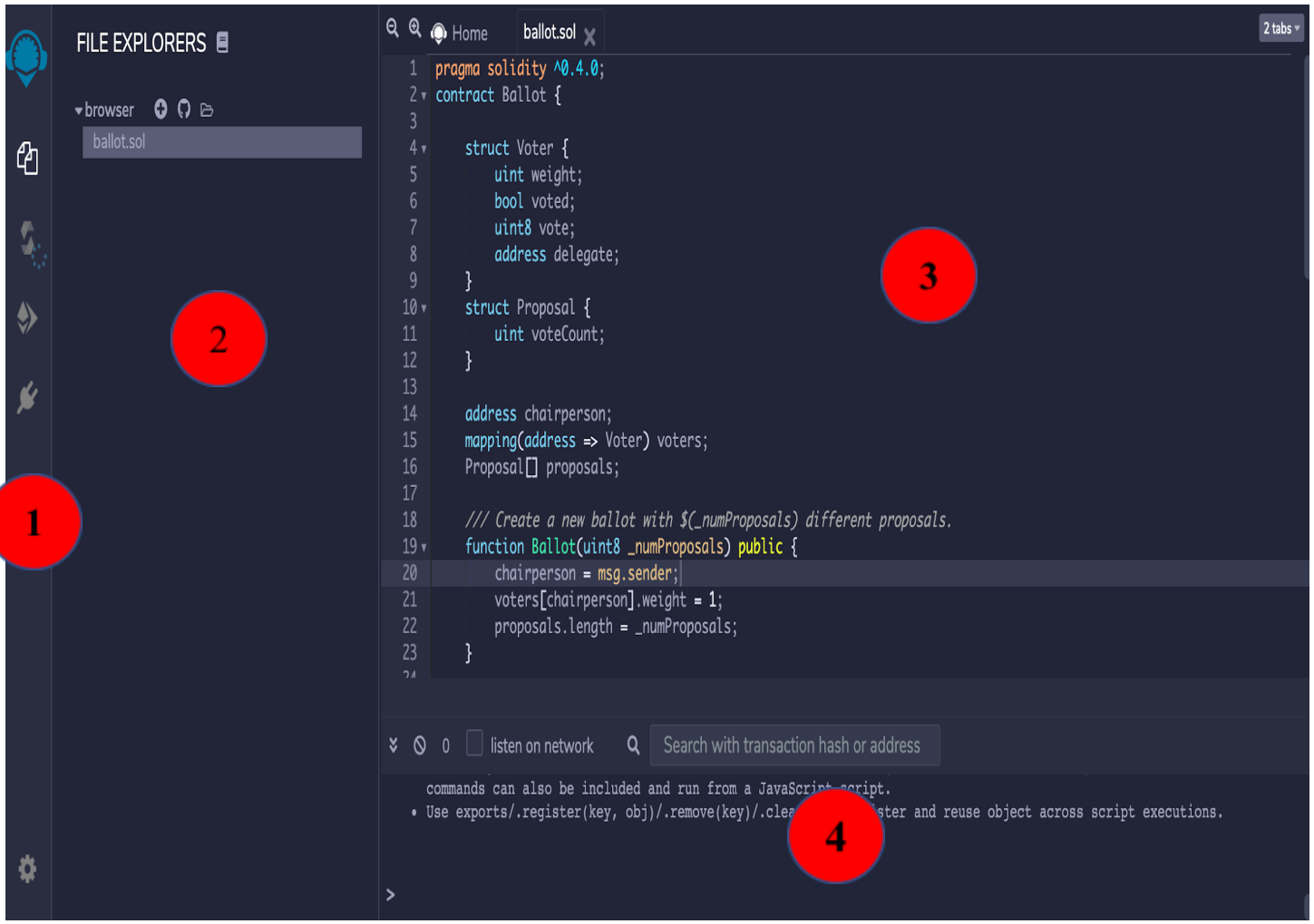
- Remix online, a development environment for Solidity
- Run on local machine by installing Solidity using NPM
- Uses Solidity's docker image
- Uses Truffle, a local development environment for writing smart contracts
- **Also try** [Ethereum Studio]

Remix Familiarisation

- The Remix compiler helps you **write**, **debug** and **run** smart contracts.
- You can run Remix IDE both **online** and **offline**.
- There are two versions available to use in your browser. One is the **latest stable** version, and the other is still in the **development** stage.
- Remix can either be opened **online** or **offline**.
- You can access both the **latest stable** version and the **alpha** version of the integrated development environment (IDE).
- By using Remix Solidity IDE, we can easily **write** and **run** contracts as well as **debug** them.

The Layout of Remix IDE

As you open **Remix Solidity online compiler** in your browser, you will see the layout consisting of four basic parts as shown in the following image.



1. **Icon panel**
2. **Swap panel**
3. **Main panel**
4. **Terminal**

In **the icon panel**, you can choose a plugin you wish to see in the **swap panel**.
The **main panel** allows you to view and edit files in multiple tabs. It highlights Solidity keywords, making it easier to grasp the syntax. The panel recompiles the code after each change and saves it a few seconds after the last one is finished.

You can see the results and run your code in the **terminal**. It shows all the important operations and transactions mined currently. You can also search for specific data and clear logs in the terminal.

Here are some of the most common plugins

Compiler	Compiles a contract

Run & Deploy	Sends a transaction
Debugger	Debugs a transaction
Analysis	Presents data of the last compilation

Using Remix Offline

Even though Remix is a Solidity browser compiler, you can use it **without** internet access as well. For that, you will need to download Remix IDE on your computer.

The latest Remix version can always be found on GitHub. After you download the .zip file (remix-21e467b1.zip), you will have to unzip it and find the index.html file. After you open the index.html file using your browser, you will be able to access Remix IDE offline.

Contract File Structure

Every smart contract file or a Solidity file is saved with the .**sol** extension.

Every .sol or smart contract file has the following main components:

- **One or more pragma statements:** The main purpose of Pragma statements is to define the Solidity version number. The Solidity compiler, i.e., solc, checks for this version number and then converts the Solidity code to the EVM bytecode accordingly. The version number is important since new functionalities are added continuously, and some old functionalities keep getting phased out.
- **One or more import statements:** Import statements are used when you want to use the functions defined in some other sol file in your sol file.
- **One or more contract definitions:** Contract is defined using the contract keyword followed by the series of functions that you write inside the curly brackets.

Solidity Programming Language

Solidity Basic Syntax

- Solidity is a high-level, statically-typed smart contract programming

- The language is similar to JavaScript.
- solc is the Solidity command line compiler.
- It is case-sensitive.
- Every line must end with a semicolon (;)
- It uses curly braces { } for delimiting the blocks of code.
- // represents a single line comment.
- /* … */ represents a multi-line (block) comment.
- /// represents a single line natspec comment and /** … */ represents a block.
- Natspec comment: Natspec is used for function declaration documentation.
- Most of the control structures such as **if, else, while, for, break, continue and return** are available.

Solidity Variables

1. State Variables:

- They are permanently stored in contract storage. Storage is not dynamically allocated.
- They are declared at the contract level.
- They are initialised at declaration, using a constructor or after contract deployment by calling setters.
- They are expensive to use; they cost gas.

2. Local Variables:

- They are declared at function level.
- If you are using the memory keyword in arrays or struct, they are allocated at runtime. Memory keywords cannot be used at contract level.

Data Types in Solidity

Following are some of the data types:

- **int**: These are signed and unsigned integers of various sizes – *uint8* to *uint256* and *int8* to *int256*.
- **bool**: Its values are *true* and *false*.
- **Fixed**: These are signed and unsigned fixed-point numbers of various sizes. They can be declared as either *fixed* or *unfixed*. Fixed point numbers are not fully supported by Solidity yet. They can be declared but cannot be assigned to or from.
- **Address**: Remember from the key-value pairs of Ethereum accounts, key was a 20-byte string, which represented the account address. It is declared as *address* and holds a 20-byte value (the size of an Ethereum address).
- **Enum**: Enums provide a way to create a user-defined type in Solidity. They are declared using *enum*.

- **Array**: Arrays can have a compile-time fixed size or can be dynamic. An array of a fixed size n and element type A is written as A[n], and an array of a dynamic size is written as A[].

1. **Fixed-size byte array**: It refers to a fixed-size byte array, which goes from 1 to 32. In other words, the size of the array can range from 1 to 32 bytes. It can be declared as *bytes1, bytes2, ..., bytes32.*
2. **Dynamic-sized byte array**: You can declare a dynamic-sized byte array with maximum size up to 256 bytes. You can declare it as *bytes*.

- **String**: String is another type of dynamic-sized array that is used for an arbitrary length string (UTF-8) data. You can declare it as *string*.
- **Mapping**: Mapping types are declared as *mapping(_KeyType => _ValueType)*. Here *_KeyType* can be almost any type, except a mapping, a dynamic-sized array, a contract, an enum and a struct. *_ValueType* can actually be any type, including mappings.
- **Struct**: A struct data type is a group of multiple data types combined together. Solidity provides a way to define new types in the form of structs.

Visibility of a State Variable

- **Public**: If a state variable is defined as public, then it can be accessed from outside the contract namespace. In other words, it can be accessed from any other contract on the network.

- **Private:** If a state variable is defined as private, then it can be accessed only inside the smart contract in which it is defined. Even if you import this contract into some other contract, the state variable with private visibility cannot be used from the contract inheriting the parent contract.

- **Internal**: A variable with internal visibility can only be used from the contract in which it is defined or the contract inheriting the properties of the parent contract. So, if a contract 'A' inherits a contract 'B', then it can use the variables in 'B' with the internal visibility type but not with the private visibility type. The variables with public visibility can be used by any contract on the network.

***** Internal is similar to private. The only difference is internal can be used in the contract that inherits this contract.**

A state variable is defined as:

<visibility type> <data type> <variable name>;

For example,

public uint votingCount;

internal mapping(address=>bytes32) sampleMapping

Function

A function is an executable unit of code within a contract. Functions create the contract interface.

There are four types of functions in Solidity:

- **View:** This function type means that the function only reads from the smart contract and will not perform any kind of writing operations. A 'view' function does not consume any amount of gas. Hence, if your function is meant to only read data from the blockchain and not write anything, then you will need to ensure that you define your function type as view. Otherwise, the function will end up using some amount of gas.
- **Pure:** This function type means that the function will neither read nor write anything on the smart contract. They are called helper functions. Math functions are a good example of pure functions. They will perform some mathematical calculations and return the final computed value. However, these functions neither read from the blockchain nor write on it.
- **Payable:** Only the functions defined as payable can take ethers as input. Hence, if you are sending ethers to smart contracts, then you must ensure that it is only to those functions that are defined as payable. Sending ethers to non-payable functions will result in failure.
- **Fallback:** If an incoming transaction call does not match any of the functions defined in a smart contract, then the call will be directed towards the fallback function. In other words, if there is a call for a function that does not exist, then the call will fall to the fallback function. There is exactly one fallback function in every contract. This function cannot have arguments and cannot return anything.

***** Function type is not mandatory.**

A function in Solidity is defined as follows:

function <function_name>(<arguments to the function>) <visibility_type> <function_type> <modifiers> returns(<return data type>)

Visibility of a Function

The visibility types for a function are the same as the visibility types for the state variables, except that there is one more visibility type available for the functions called external.
External: A function with the visibility type as external can only be accessed from outside the contract. A function with external visibility cannot be called from the contract in which it is defined.

Function Modifiers:

- They **test a condition before calling a function**. The function will be called only if the condition evaluates to true.
- Using function modifiers, we **avoid redundant code and possible errors**.
- They are contract properties and are inherited.
- They **do not return** and use only **require ()** and **semicolon (;).**
- They are defined using the **modifier** keyword.

A modifier is a special kind of function that contains certain user-defined conditions. When a modifier is used in a function definition, all the conditions inside the modifier are checked, and if all the conditions are satisfied or are equal to true, only then will the function execution move forward.

A modifier acts as a pre-check to the function execution. It always ends with '_;' before the closing parenthesis. '_;' is replaced with the actual function body when the modifier is used.

Events:

- Each transaction has an attached receipt which contains zero or more log entries that represent the result of events fired from that smart contract.
- Log entries represent the transaction outcome and are saved on the blockchain. They are associated with the contract address.
- Log entries are historical data and are not used for consensus but are verified for future references.
- Transaction receipt hashes are stored inside the block header.
- An event is generated in the function body which wants to generate the log.
- Events can be used inside the setter and not getter functions.
- DApps and web applications listen for events and can execute a 'JS callback function' that updates the UI according to the transaction outcome using web3.

An event in Solidity is defined as follows:

event <event_name> (<event_parameters>)

After you have defined an event, you can call it or trigger it from inside any function you wish. You can call an event using the *emit* keyword in the following manner:

emit <event_name> (<event_arguments>)

- An event can be indexed so that it is searchable using an indexed keyword in a parameter.
- This is useful in case someone wants to search the history and filter the events in a particular index.
- Indexed parameters are stored inside the logs in a special kind of data structure called **'topics'.**

Global Functions

These are globally available functions (or in-built functions) in Solidity.

Global functions are grouped into four major contexts as follows:

- Address
- Block
- Transaction
- Message

The in-built functions defined in the address context are as follows:

- **balance:** The balance function returns the balance of the address in Wei. For an address mycontract, the function call is defined as '*mycontract.balance;*'. This will return the balance of mycontract in Wei. The return type is uint256.
- **transfer():** The transfer function sends the given amount of Wei from the current account to the mentioned address. If you want to send x amount of Wei from the current account to an address mycontract, then the function call will be defined as '*mycontract.transfer(uint256 x);*'. The problem with the transfer function is that an error during the transaction will cause it to fail.
- **send():** The send function sends the given amount of Wei from the current account to the mentioned address. If you want to send x amount of Wei from the current account to an address mycontract, then the function call will be defined as '*mycontract.send(uint256 x);*'. The return type of the send function is bool. Moreover, whenever a transaction encounters an error, the send function returns false. Based on this, appropriate action can be taken. This is what differentiates the send function from the transfer function.
- **call()**, **staticcall()** and **delegatecall():** These are low-level functions which do not go through the checks by the Solidity compiler and, hence, it is advised not to use them unless absolutely necessary. The call and staticcall functions work in a similar manner as the transfer or the send functions. However, delegatecall works differently. If A invokes B which makes a delegatecall to C, then the msg.sender in the delegatecall will be A and not B. This way, we can preserve the original sender of the message. All these low-level functions take bytes as input and the return type is a combination of bool and bytes data types.

The in-built functions defined in the block context are as follows:

- **block.coinbase:** It returns the address of the miner than mined the current block.
- **block.difficulty:** It returns the difficulty at the time when the current block was mined.
- **block.timestamp:** It returns the timestamp at which the current block was mined**.**
- **block.gaslimit:** It returns the total gas limit of all the transactions mined in the current block.

- **block.number:** It returns the number of the newest block in the blockchain.

The in-built functions defined in the transaction context are as follows:

There are functions in the global namespace which are used mainly to provide information about the transactions in the network. There are two such main functions defined as part of the transaction context. These are as follows:

- **tx.gasprice:** It returns the gas price of the transaction sent by the sender as part of the transaction.
- **tx.origin:** It returns the address of the original sender of the transaction.

The in-built functions defined in the message context are as follows:

Some global functions are used to capture the properties of the messages. The main functions in the message context are as follows:

- **msg.value:** This function returns the number of Wei that was sent with the message or the transaction.
- **msg.sender:** This function returns the immediate sender of the message or the transaction. Unlike tx.origin, msg.sender returns the address of the previous account in the flow of the message. If A sends the message to B and B sends it to C, after which if C calls msg.sender on that message, it will receive the address of B as the return value.
- **msg.gasleft:** This function returns the remaining gas for the transaction. If an account feels that the remaining gas is inadequate or insufficient for a transaction to complete, then it will fail the transaction

Apart from the functions belonging to the four contexts that you saw, there are two important global functions in Solidity:

- **now():** This function returns the timestamp when the last block in the blockchain was created. Since the block creation rate in Ethereum is approximately 15 seconds, the timestamp returned will be approximately 15-30 seconds prior to the current time. The timestamp returned by now() is the number of milliseconds since the Unix epoch time.
- **selfdestruct():** As the name suggests, this function is used by a contract to destroy or kill itself. You need to pass an address as argument to this function, and all the balance ethers present in the contract account will be transferred to the account with the address passed as an arguement.

Inheritance:

- In object-oriented programming, inheritance is a feature that represents the 'is a' relationship between different classes or contracts.
- Inheritance allows a class or a contract to possess the same behaviour (functions and variables) as another class or contract and allows modifying that behaviour as per the need. In Solidity, a contract that is inherited is called the parent or the base contract, and a contract that inherits is called the child or the derived contract.
- All public and internal scoped functions and state variables defined in the parent contract are available for use in the child contracts.
- Inheritance is a very important concept because it brings the code reusability feature to the table.
- To inherit a contract in Solidity, you need to perform the following steps in your child contract:
- A contract can inherit from other contracts which are known as base contracts.
- When a contract inherits from multiple contracts, only a single contract is created on the blockchain, and the code from all the base contracts is copied into the created contract.
- The general inheritance system is very similar to Python's, especially concerning multiple inheritance.
- Solidity supports multiple inheritance including polymorphism. Multiple inheritance introduces problems like the 'diamond problem'.
- All function calls are virtual, which means that the most derived function is called, except when the contract name is explicitly given.
- When deploying a derived contract, the base contract's constructor is automatically called.
- 'is' keyword is used when declaring a new derived contract.

Use the import statement to import the parent contract. The syntax for the same is as follows:

import '<path to the parent contract file>';

After importing the file, you need to make the current contract an inherited contract. You can do this using the 'is' keyword. The '**is**' keyword is used to inherit the base contract in the derived contract. The syntax for the same is as follows:

contract <contract_name> is <contract_name of the parent class>{<contract code>}

Lifecycle of a Smart Contract

There are two major elements that define the birth and death of a smart contract in Solidity. They are as follows:

- **Constructor:** A constructor is a function declared with the constructor keyword. It is called when a contract is installed on the network. It is executed only once. Moreover, it is optional to use the constructor function. After the constructor has executed, the final code of the contract is deployed to the blockchain. The code deployed does not include the constructor code or the internal functions only called from the constructor. Other than this, the code deployed includes everything defined inside the contract. A constructor is defined as follows:

constructor(<parameters>) <visibility_type>

{ <code that you want to execute at time of contract deployment/installation> }

Since the constructor is the first ever function to be called in a smart contract, it is called part of the contract creation call, which can be made only by an externally owned account (EOA). Hence, it is advisable to keep the visibility type of the contract public. In fact, Solidity only allows the visibility type of a constructor to be public or internal. A constructor cannot be private or external.

- **Selfdestruct:** When called, the selfdestruct function kills or deletes the entire smart contract from the blockchain. However, this does not delete the contract account, and the contract account on which this contract code was deployed becomes a blank account. You can define a selfdestruct function as follows:

selfdestruct(<address of the account to which the balance ethers will get transferred to>) ;

You need to pass an address of an account as a parameter to the selfdestruct function. This address is the address of the account to which all the balance ethers in the current contract account will be transferred.

Deployment and Testing of Smart Contracts

Now, compile the contract code and then deploy and run the contract on a sample blockchain inside the Remix environment. Before moving on, ensure you have the following plugins activated in the Remix IDE:

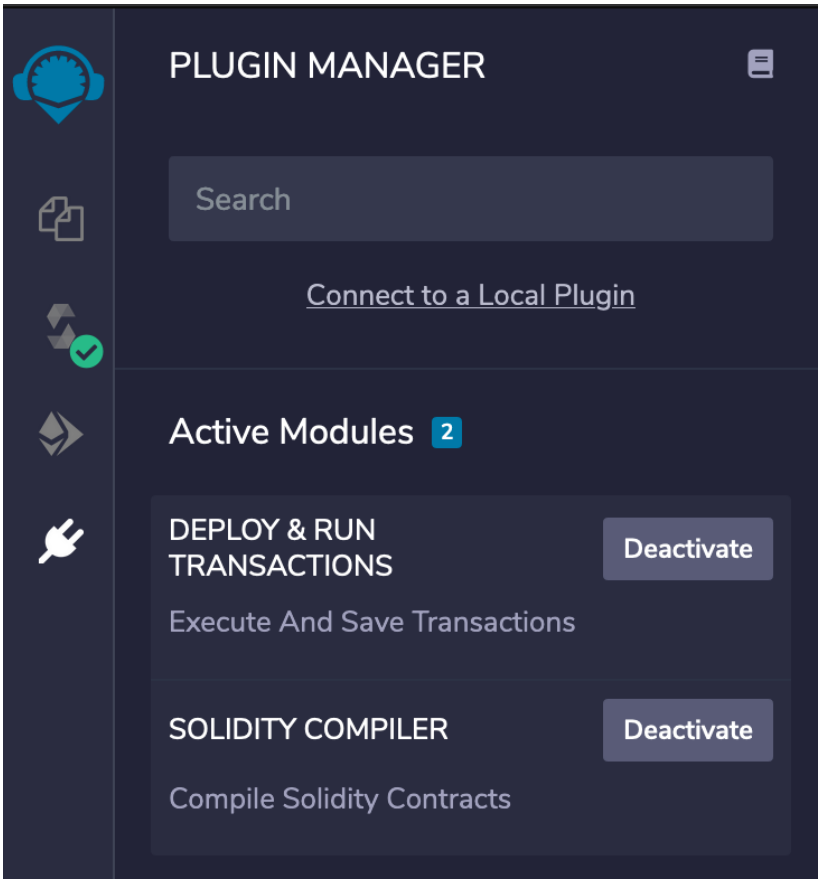
- Solidity compiler
- Deploy and run transactions

Remix icons

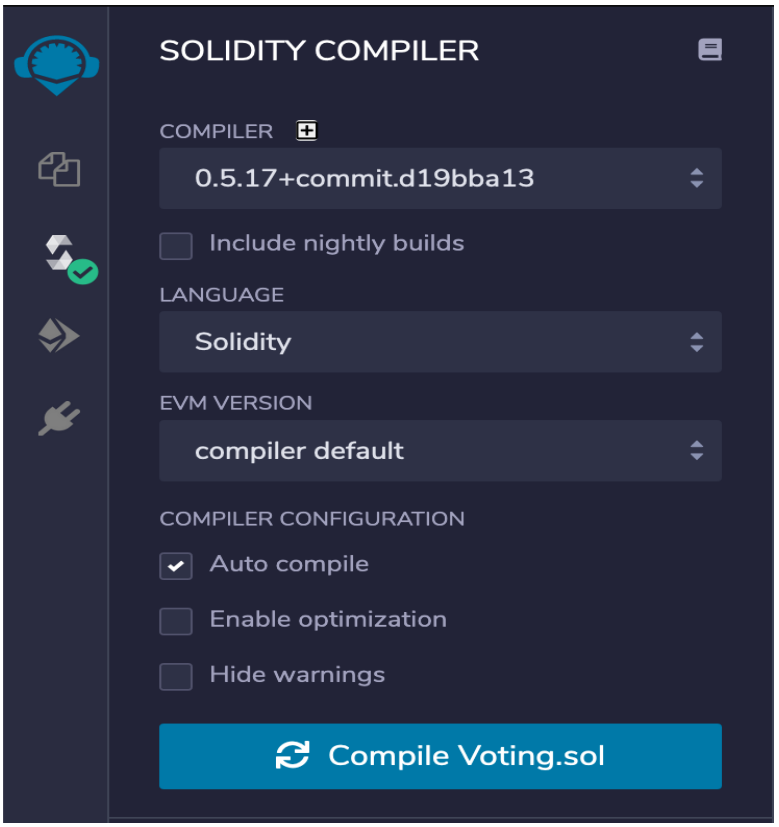
- Plugging

 Compilation

- Deploy

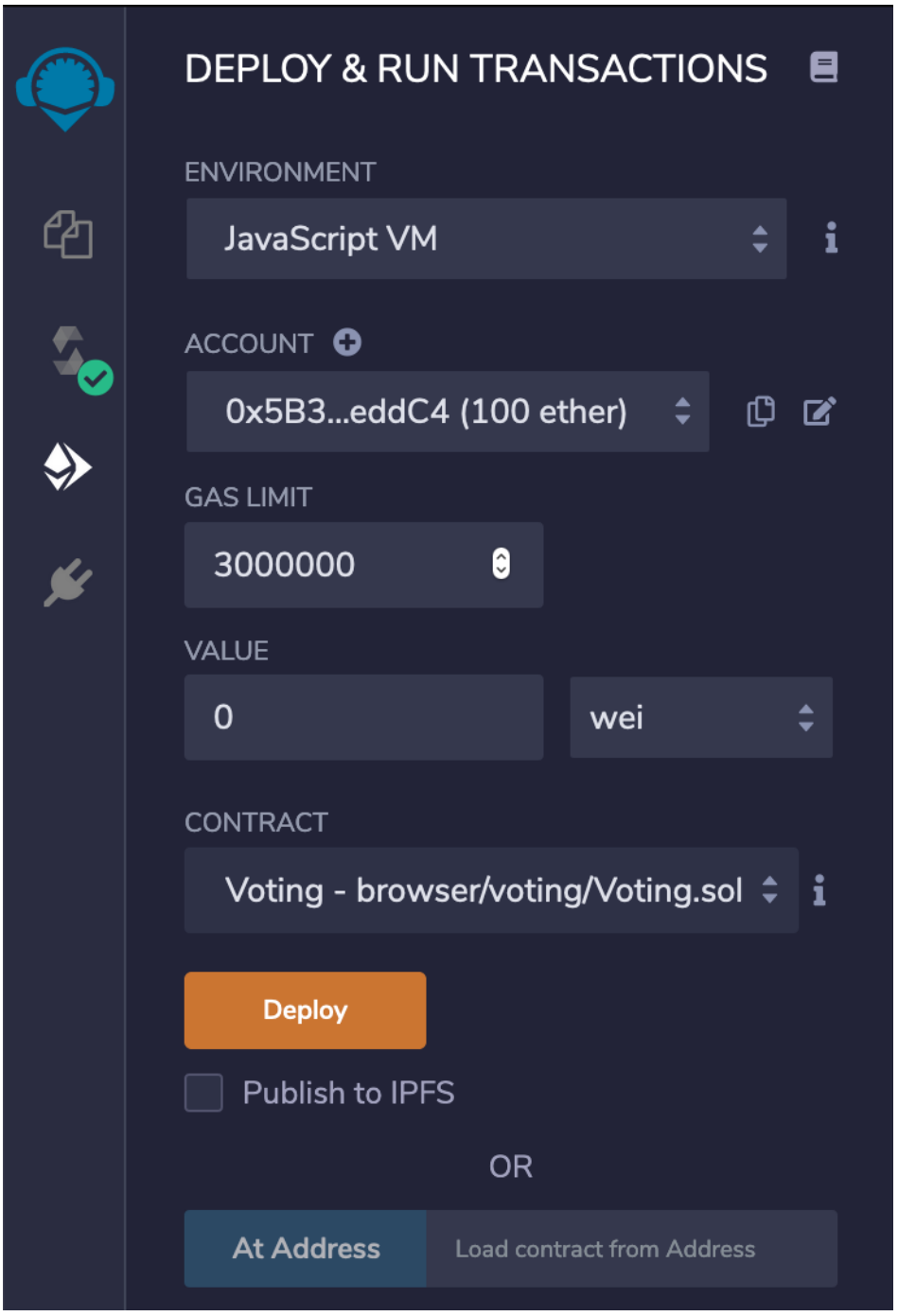


In Remix, select the 'Compile' tab in the top right hand side of the screen, and start the compiler by selecting the 'Start to Compile' option. You can also select 'Auto Compile'.



If everything is ok, you should see a **green label** called ‘Voting’ with the name of your contract. This indicates the compilation was successful.

In Remix, select the ’Deploy’ tab. Within the Environment dropdown section, select the ’JavaScript VM’ option.

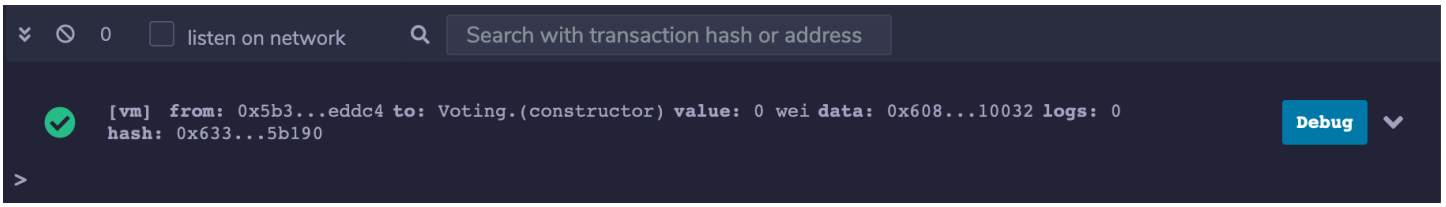


The ‘JavaScript VM’ option runs a JavaScript VM blockchain within the browser. This allows you to deploy and send transactions to a blockchain within the Remix IDE in the browser.

With the JavaScript VM environment option selected, click the ’Deploy’ button.

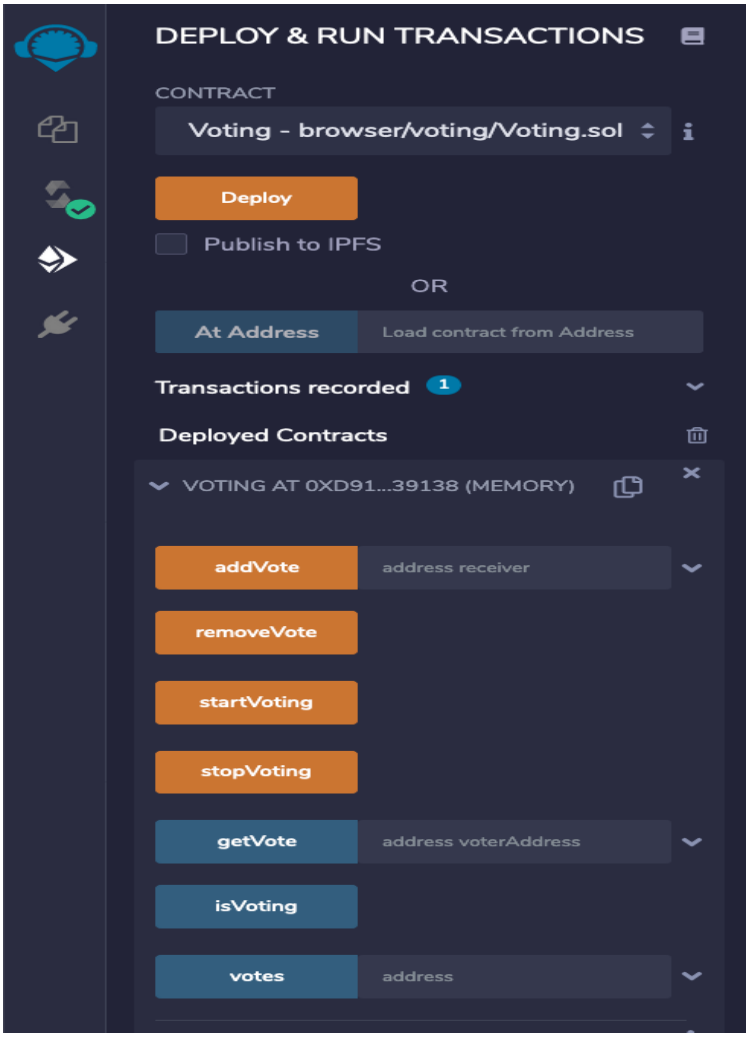
This button executes a transaction to deploy the contract to the local blockchain environment running in the browser. We will talk more about contract creation transactions later on in the series.

Within the Remix IDE console, which is located directly below the editor panel, you will see the log output of the contract creation transaction.



The ‘green’ tick indicates that the transaction itself was successful.

Under the ‘Deployed Contracts’ section, we see a list of functions which can be invoked on the deployed smart contract.



The purpose of compiling and running your code in an environment like Remix is so that your code is battle-tested. In other words, you would not want your code to be deployed on a real world blockchain only to find that the contract code is incorrect. There could be compilation errors, runtime errors, or some logical mistakes in your contract code. Hence, it is highly advisable to test your code on a sample blockchain inside Remix or any other similar development environment before deploying your contract on a real blockchain.

Error Handling in Solidity

There are three methods that constitute the error-handling process in Solidity. These are as follows:

- **require():**
  require() is used to validate certain conditions before further execution of a function. It takes two parameters as input. The first parameter is the condition that you want to validate, and the second parameter is the message that will be passed back to the caller if the condition fails. If the condition is satisfied, then the execution of the function continues and the execution jumps to the next statement. However, if the condition fails, then the function execution is terminated, and the message (the second parameter) is displayed in the logs. The second parameter, however, is optional. require() will work even if you only pass the parameter, that is, the condition to be checked. The require() statement is defined as follows:

require(<condition to be validated> , <message to be displayed if the condition fails>);

- **assert():**
  The assert function, like require, is a convenience function that checks for conditions. If a condition fails, then the function execution is terminated with an error message. assert() takes only one parameter as input. You pass a condition to assert(), and if the condition is true, then the function execution continues and it jumps to the next statement in the function. The assert() statement is defined as follows:

assert(<condition to be checked/validated>);

- **revert():**
  The revert function can be used to flag an error and revert the current call. You can also provide a message containing details about the error, and the message will be passed back to the caller. However, the message, like in require(), is an optional parameter. revert() causes the EVM to revert all the changes made to the state, and things return to the initial state or the state before the function call was made. The reason for reverting is that there is no safe way to continue the execution because something unexpected happened. This is important as it helps in saving gas. Since the function execution stops after the revert() statement, the remaining gas is also returned back to the user. If you do not use the revert() statement and some error occurs, then the entire gas is lost. However, using revert() does not return the consumed gas. The gas that is consumed is consumed and cannot be returned.

In both require() and assert(), if the condition fails, then the function call is reverted to the initial state or the state before the call. Since you want to retain the atomicity of the transactions, the safest thing to do is to revert all the changes made as part of the function and return to the initial state. As operations are performed inside a function, a substate is maintained. Now, if the condition in require() or assert() fails, or if the revert() method is called, then the substate vanishes and things return to the initial state. However, if the function executes till completion, then the final substate is considered the finalised state or the new state. This ensures that the atomicity of the Ethereum states is maintained. If a function fails, things return to the initial state, or if a function executes successfully, then things move to a new state.

Difference between require() and assert()

require()

- require() is used to check the conditions on the input parameters that act as input to the function.
- require() is used at the start of the function and acts as a pre-check to the function.
- In require(), if the condition fails, then the unused or the remaining gas is refunded back to the user.

assert()

- assert() is generally used to check for any internal errors.
- The assert statement is used in between the operation statements inside a function
- assert() is used to check whether a certain thing is leading to an error or not.
- The unused or remaining gas is not returned back in case an assert() exception occurs.

Use of require() and assert()

assert() is used to indicate that this condition [which is passed inside assert()] should never fail. If, however, this condition fails, then it means that there is something very wrong with the code. If an assert type error is raised, then the source code analyser knows that there is something extremely wrong and can possibly detect a logical error in the system by discovering any situation in which the assert statement might not be true.

require() is good to validate the input and is generally useful for finding whether the user who called the function passed some incorrect inputs or not. assert(), on the other hand, is meant

to indicate something that should never be false under any circumstances. Knowing this, a source code analyser can debug the code by following the assert statements in the source code and detecting the situation where those statements might have failed. It is hence said that one should write his/her contract such that assert exceptions never occur.

Accessing Other Smart Contracts

The contract code is stored as the EVM bytecode on a contract account in the blockchain. When you inherit a contract, the Solidity compiler copies the bytecode of the base or the parent contract into the bytecode of the child or the derived contract.

The bytecode present on a contract account contains the bytecode for the main contract as well as for all the other contracts that are imported into that contract.

If you want to access a smart contract that is already deployed on the network, you can do it in either of the following two ways:

- Create a new instance of the deployed smart contract
- Use an existing instance of the deployed smart contract that you want to access

Both these approaches work only when you have the Solidity file for the contract that you are trying to get access.

Method - 1

When you create a new instance for a smart contract that is already deployed on the blockchain, a new contract account is created, which contains the same contract bytecode as the original contract account.

Now, you access the functions of the contract using the address of the new contract account. You can define a new instance as follows:

<deployed_contract_name> <variable_name>;

<variable_name> = new <deployed_contract_name>(<parameters to the constructor of the deployed smart contract>);

Here, the **<variable_name>** stores the address to the new contract account. The new contract account will contain the bytecode of the **<deployed_contract_name>.** Now, you can call any function of the contract using the following statement:

<variable_name>.<function name and function parameters>.

Method - 2

You can access another contract using an already existing contract instance. To do this, you need to define a variable of that contract, and store the address of the contract account that contains the contract bytecode that is to be accessed in that variable. You can use an existing instance as follows:

<deployed_contract_name> <variable_name>;

<variable_name> = <deployed_contract_name>(address of an already existing contract account that contains the contract code to be accessed);

Here, the **<variable_name>** stores the address to the existing contract account. Now, you can call any function of the contract using the following statement:

<variable_name>.<function name and function parameters>.

Accessing a contract whose Solidity file you do not have

This is generally the case when you try to access a smart contract that is deployed by someone else. Now, you will have to use low-level functions, such as call() or delegatecall(), to access contracts whose Solidity files you do not have. For accessing a smart contract, you need to know the address of the contract account that stores the bytecode for the contract that you are trying to access. Using this address and the low-level functions, you can access this contract. To make a function call on an already deployed contract, you can use the call or delegatecall functions as follows:

<address>.call(<function name and parameters>);

<address>.delegatecall(<function name and parameters>);

*** **It is highly advisable to not access a contract that you have not created**. **Ethereum highly discourages accessing a smart contract that you have not deployed.**

Additional Notes

Struct

- A struct is a collection of key->value pairs similar to a mapping, but the values can have different types.
- A struct introduces a new complex data type that is composed of elementary data types.
- We use structs to represent a singular thing that has properties such as a car, a person and a request and we use mappings to represent a collection of things such as a collection of cars and requests.
- A struct is saved in storage and, if declared inside a function, it references storage by default.

Example:

```
struct Car {
string brand;
uint built_year;
uint value;
}
```

Mappings

- It is a data structure that holds key->value pairs and is similar to Python Dictionaries, JS objects or Java HashMaps.
- All keys **must have the same type** and all values must have the same type.
- The keys cannot be of mapping, dynamic array, enum or struct types. The values can be of any type including mapping.
- Mapping **is always saved in storage**; it is a state variable. Mappings declared inside functions are also saved in storage.
- The advantage of mapping is that the lookup time is constant no matter what the size. Arrays have a linear search time.
- A **mapping is not iterable**. We cannot iterate through a mapping using a for loop.
- **Keys are not saved into mapping** (its hash table data structure). To get a value from mapping, we provide a key, which gets passed through a hashing function that outputs a predetermined index. The index in turn returns the corresponding value from the mapping.
- If we want the value of a **non-existing** key, we get a **default value.**

Transactions, Blocks and Mining

- Transactions are the heart of the Ethereum blockchain. Interacting with the Ethereum blockchain means executing transactions that update the blockchain state.

The term ‘transaction’ is used in Ethereum to refer to the signed data package that stores a message to be sent from an EOA to another account on the blockchain.
Contracts have the ability to send ‘messages’ to other contracts. An EOA produces a transaction and a contract produces a message.

Life Cycle of an Ethereum Transaction

1. Client constructs the raw transaction object.
2. Client signs the transaction and validates it locally.
3. Transaction is broadcasted to the network by the Ethereum client and a transaction ID (txid) is returned.
4. The miner node accepts the transaction. The Ethereum network has a mix of miner nodes and non-miner nodes. The transaction is added to the transaction pool and waits there to be validated by the miner.
Transactions in the pool are sorted by gas price.