

Ethereum Smart Contract Development

Contents

Course Overview	5
Introduction to Ethereum SC development.....	6
The Ethereum Network	6
Nodes/Clients:	6
ETH:	8
Blockchain Basics	8
Ethereum Accounts	13
Transactions and Signatures.....	14
Ether, Wei and Gas.....	17
Smart Contracts and Solidity.....	19
Setting up Metamask	21
Writing, compiling, deploying and debugging smart contracts on Remix.....	24
Home Page:	24
File Explorer:.....	25
Compiler:.....	26
Deploy & Run Transactions:	27
Environment:	28
Hello World Smart Contract	29
Setting up a local development environment	30
Compiling, testing and deploying smart contracts with Hardhat	31
What is Hardhat?	31
Creating a Hardhat project:	31
Configuring Hardhat:	32
Compiling a Hardhat project	32
Deploying smart contracts with Hardhat.....	33
The Hardhat Network:	33
Debugging with Hardhat	34
Solidity	35
Features and basic layout of a solidity smart contract:	35
Solidity Types	37

Visibility of State variables	37
Value Types.....	37
Integers and unsigned integers	37
Boolean.....	38
Address.....	38
Fixed size byte arrays	39
Enums.....	39
Reference Types	40
Strings and Bytes:.....	41
Arrays:	41
Struct	41
Mapping	42
Special variables.....	42
msg	42
block.....	42
Error Handling.....	43
Functions	44
Function visibility	45
Function overloading.....	46
Special Functions	46
getter functions	46
constructor	46
selfdestruct	46
receive	47
fallback	47
Modifiers	48
Inheritance	48
Importing Files	49
Events	49
Libraries	51
Abstract Contract and Interface.....	52
Abstract contracts	52
Interfaces	52
Voting smart contract	53
Using smart contracts from a web frontend - Ethers.js	54

Provider	55
API Provider.....	55
DefaultProvider	55
Signers	56
Contract	57
ContractFactory	58
Types	58
Utilities	59
Fragments	59
Interface.....	60
Addresses	62
BigNumber	62
Byte Manipulation.....	63
Constants	63
Display Logic and Input.....	63
Hashing Algorithms	64
Strings.....	64
Transactions	65
The Metamask RPC API.....	66
Ethereum provider API	66
Errors	67
React DAPP Project	68
Requirements:.....	68
Exercise:	69
ERC20 Tokens and OpenZeppelin.....	70
ERC20 Tokens	70
OpenZeppelin	71
ERC20 Token Project.....	72
Requirements:.....	72
Exercise:	72
Testing Smart Contracts	73
Basic structure of a test file	73
Hardhat Chai Matchers.....	74
Hardhat Network Helpers.....	75
Fixtures:.....	76

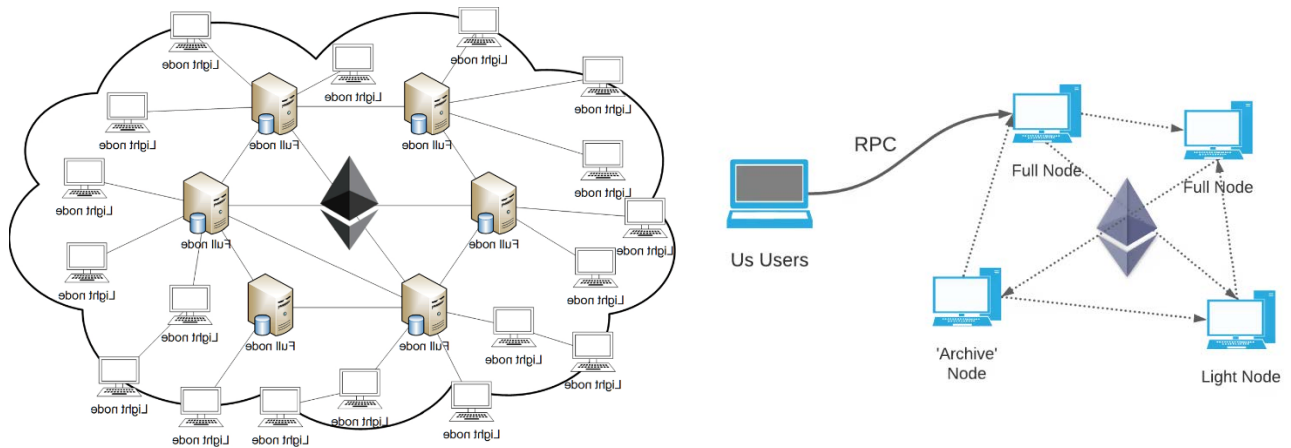
Requirements:.....	77
Exercise:	77
NFT's and ERC721 Tokens.....	78
What are NFT's.....	78
ERC721 Tokens	78
IERC721:.....	78
ERC721URIStorage	80
ERC1155	80
What is IPFS:	80
What is Pinata.....	80
NFT Minter Project	82
Requirements:.....	82
Exercise:	83
Protecting your code against a reentrancy attack	84
Reentrancy Attack Project.....	85
Requirements:.....	85
Exercise:	85
DAO - Decentralized Autonomous Organization	86
What is a DAO	86
How does a DAO work?	86
Project Contracts:	87
Governance token - ERC20Votes:	87
Treasury Contract:	88
Timelock Contract - TimelockController:.....	88
Governor Contract:.....	90
Creating a proposal:	91
The DAO Project.....	92
Requirements:.....	92

Course Overview

- The basics: The Ethereum Network, Blockchain, Ethereum accounts, transactions, ETH, wei & gas...
- Metamask: Setup, configuration...
- Remix: compiling, deploying and debugging smart contracts
- Hello World: Our first SC
- Hardhat: Setting up a local development environment to compile, test and deploy SC's
- Solidity: Types (struct, mappings...), functions (receive, fallback...), modifiers, events, error handling, inheritance...
- Voting smart contract
- Creating a React web front-end using Ethers.js: Provider, signer, contract, utils
- The Metamask RPC API
- Creating an ERC20 token using OpenZeppelin
- Testing smart contracts in Hardhat using Mocha and Chai
- Creating ERC721 NFT's with OpenZeppelin, IPFS and Pinata
- Protecting your code against a reentrancy attack
- Creating a Decentralized Autonomous Organization (DAO) - Governance token, treasury, timelock and governor contract

Introduction to Ethereum SC development

The Ethereum Network



Vitalik Buterin started working on Ethereum in 2013 and the first production release was launched in 2015. Ethereum is an open-source, public, decentralized, permissionless and immutable blockchain that allows the deployment and execution of smart contract code.

Most smart contracts are written in a programming language called Solidity which is Turing-complete (Turing Complete refers to a machine that, given enough time and memory along with the necessary instructions, can solve any computational problem). Those smart contracts are compiled into byte code, which is executed on the Ethereum Virtual Machine - EVM

Ethereum is a P2P network that consists of about 1800-2000 nodes. Each node has a copy of the latest up-to-date blockchain ledger and each of those nodes (computers) is running a software program that executes the various features of the Ethereum protocol. Those programs are called Ethereum clients and the most commonly used ones are Geth and Parity.

In such a P2P network 2 or more computers (clients) are connected and share their resources without going to a centralized source.

Nodes/Clients:

Nodes act as a gateway making it possible for users and applications to get access to the information stored on the blockchain and interact with the network. The nodes are constantly receiving and sending out data to each other, as they work together to reach a consensus of transaction validity.

When a transaction is sent to the Ethereum network, it is sent to an Ethereum node, which checks the transaction data and also broadcasts it to the nodes it is connected to in the

network. They also check the data and broadcast it to the Ethereum nodes they are connected to, and so on and so forth until all the nodes (or majority) all have the same information.

There are 3 types of nodes in the network:

Full Nodes:

Full nodes store the current and most recent blockchain states (up to the last 128 blocks) and participate in validating newly added blocks (can receive transactions). They can process transactions, execute smart contracts, and query/serve blockchain data. Full nodes prune blockchain data and only keep the minimal data necessary to verify the network's state.

Smart contracts can also be deployed to an Ethereum full node directly. Running a full node requires ~ 700 GB of disk space

Archive Nodes:

Archive nodes can store the complete historical data for the blockchain and serve it on request. These are different from full nodes that only store the recent blockchain state (last 128 blocks) and light nodes that primarily request data from full nodes.

An Ethereum archive node is a full node with the capacity to store the entire blockchain history. An archive node requires ~10 TB of disk space.

Light Nodes:

Light clients only store block headers, giving them access to minimal blockchain data (e.g., block timestamp, balance of an account...). However, they can also interface with full nodes to get additional data.

It can hold and retrieve information about a singular account (wallet app), it can also send transactions. The light nodes do not participate in consensus (i.e. they cannot be miners/validators), but they can access the Ethereum blockchain with the same functionality as a full node.

Running a light node requires ~ 400 MB of disk space

Key characteristics of Ethereum Networks:

- Used to transfer money, store data, execute smart contract functionalities...
- There are various different networks: Mainnet, test networks and private networks
- A network consists of several nodes that communicate in a P2P network
- Each node is a computer running an Ethereum client: Geth, Parity...
- Anyone can participate and run a node
- There are full nodes and lightweight nodes

- Full nodes store a complete copy of the blockchain ledger, which contains all transactions that have ever taken place

Connecting to the network:

In order to obtain information like account balances, details about specific transactions or if we want to create our own transactions in order to transfer money or execute specific smart contract functions, we need to connect with the Ethereum network.

There are different possibilities:

- Using a DAPP like Metamask - for consumers
- Using a library like web3.js or ethers.js - for developers
- Consulting a blockchain explorer - to retrieve details on specific transactions
- Using a third party provider to avoid running our own node - Alchemy, Infura

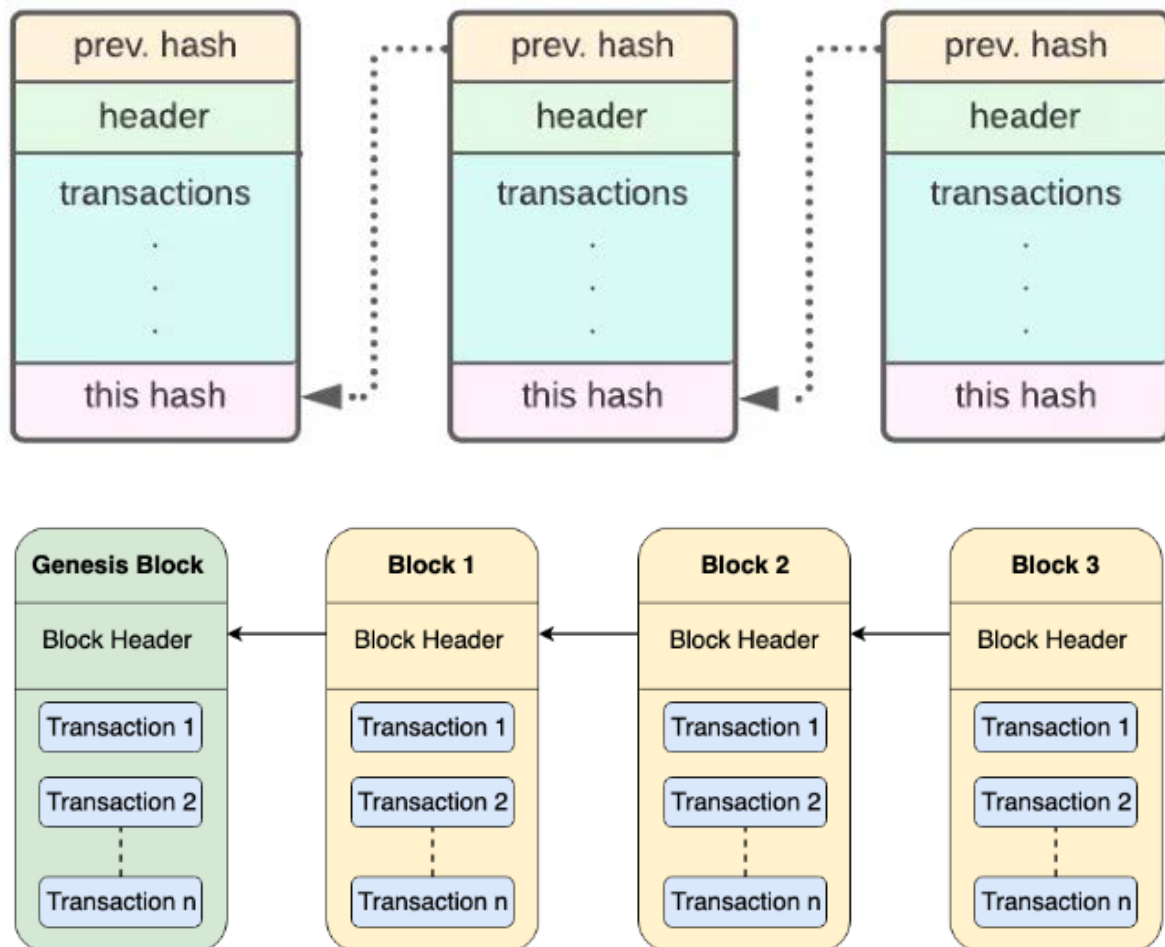
ETH:

Ether - ETH is the currency that is used on the Ethereum network. 1 ETH can be divided into smaller units - the smallest value is called wei and 1 ETH contains 10^{18} wei. Also, whenever you perform a transaction on the Ethereum network or when you execute code in your smart contract that changes state, a small fee needs to be paid. The unit of that fee is called gas. The amount of gas required depends on the transaction and the price per unit of gas (in Gwei) depends on the current state of the network.

Blockchain Basics

As the name indicates, a Blockchain is simply a chain of connected blocks where each block contains a varying number of transactions plus some additional information in the block header.

Compared to traditional centralized systems, an open, public, permissionless blockchain allows to send transactions (and therefore change data on the blockchain ledger) without a middleman - no one can block you, no one can modify your data...



About every 15 seconds a new block is generated by miners using a PoW consensus algorithm, which will be replaced by a PoS algorithm in the near future. Each block contains a block header and transactions.

Each block can hold only a limited number of transactions that number depends on the size of the transactions. Each block has a limit of 8 million gas - this means, the total amount of gas consumed by all transactions in the block cannot exceed 8 million units of gas.

The block header contains information about the block such as the timestamp when the block was created, the hash of the previous or parent block (and that is how the individual blocks are connected), a field called the Nonce which contains a specific number found by the miner to solve the cryptographic puzzle and some other fields...

Hashing:

We said, each block contains the hash of the previous block and that is a very important concept of blockchains. Now, what is a hash?

A hash is a digital fingerprint of a specific input, which could be a word, a document, an mp3 file or any other source of digital information. No matter, what input we are providing, we will always get an output or digest of a specific length. The same input will always generate the exactly same output. If we modify the input only slightly (for example, changing one letter) we will get a completely different output. Also, a hash function is a one-way function: we can generate an output from a specific input, but we cannot re-create the input from the hash.

Solidity uses the keccak-256 hashing function that generates a 32 byte (64 character) output

Example hashes:

Hello => 06b3dfaec148fb1bb2b066f10ec285e7c9bf402ab32aa78a5d38e34566810cd2

Hello1 => f78fd070e76f73c4e7282620a7ab5ba58dc4f724d5dfd45d97ec4e01f817ce9d

A hash function has the following properties:

- It is deterministic - the same input always generates the same hash
- It is quick to compute
- It is a one-way function - we cannot generate the input from the hash
- A minor change of the input value creates a completely different hash
- Two different input values will never generate the same hash

So, as we already mentioned, each block contains the hash of its previous block. This means, if someone would try to change the information in a specific block, he would also have to change all subsequent blocks and that makes it very difficult to change and manipulate the information in the blockchain ledger.

Let's take a look at an example:

<https://andersbrownworth.com/blockchain/block>

Block

Block: # 1

Nonce: 72608

Data:

Hash: 0000f727854b50bb95c054b39c1fe5c92e5ebcfa4bc5dc279f56aa96a365e5a

Mine

This is a simplified model of a blockchain. The Ethereum blockchain works slightly different, but it is nevertheless a great way to demonstrate the core concepts of a blockchain.

In this model, we increase the Nonce field (1,2,3,4...) until we find a hash with 4 leading zeroes (this is the required difficulty for the miner)

Blockchain

Blockchain

Block: # 1

Nonce: 11736

Data:

Prev: 00

Hash: 0000157830764259d382017091a36d2060900e2c203567748f46833fe9297cf

Mine

Block: # 2

Nonce: 25230

Data:

Prev: 0000157830764259d382017091a36d2060900e2c203567748f46833fe9297cf

Hash: 000012fa90916eb9078f8d90870646977ae32e54f53460d84452c0af0843c19

Mine

If we modify the data in Block 1, the hash changes and the block will no longer be valid (it does not contain 4 leading zeroes). But, as each block is connected to its previous one, the following blocks will also be invalidated and all those blocks need to be re-mined, which takes an enormous amount of computer resources and would nearly be impossible.

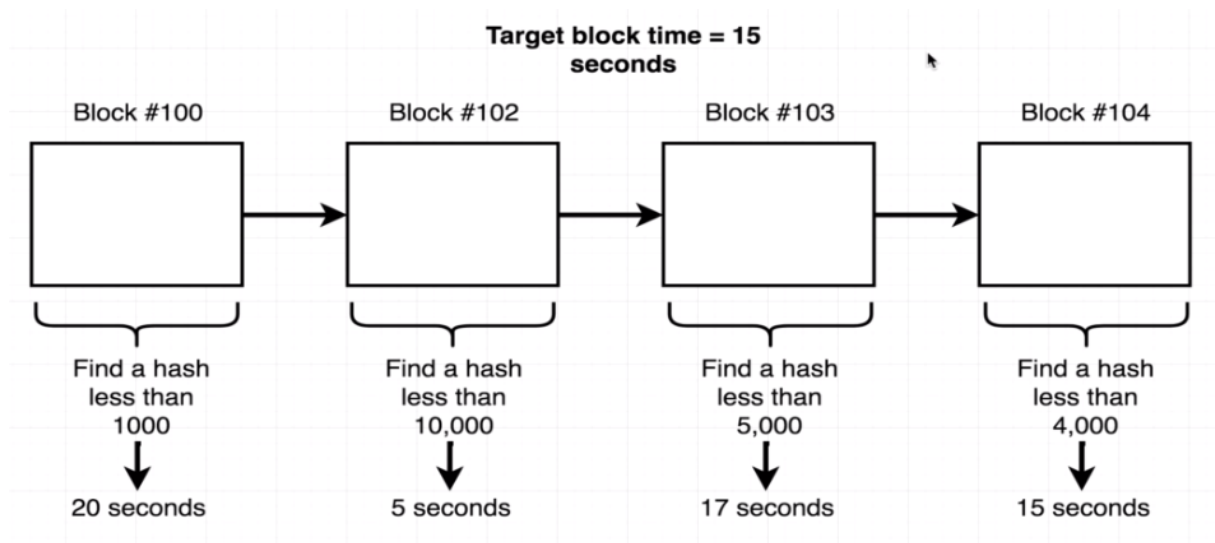
For Ethereum, the algorithm works slightly different:

The Nonce is a counter variable that's increased by 1 each iteration. We add the Nonce to our data and create a hash, then we convert that hash to a base 10 number and verify if it is

smaller than a specific target number - the desired target number is defined by the current difficulty level which changes periodically.

As soon as a miner finds that target number, a new block is added at the end of the blockchain. The time it takes to mine a new block should be around 15 seconds - if it is higher, the difficulty will be decreased, if it is lower, the difficulty will increase.

Data	+	Nonce	=	Output Hash	Output hash as a base 10 number	Is this less than 1000?
'Hi There'		0		a23042b2e	178917215	no
'Hi There'		1		cbc1491	29589283	no
'Hi There'		2		0ca24258	94869869	no
'Hi There'		3		d9eed91	13938166	no
'Hi There'		4		1488baec	419386918	no
'Hi There'		5		0077bbb	100	yes



Ethereum Accounts

There are 2 different types of accounts: Externally owned accounts (EOA) and contract accounts. Both account types can hold balances of Ether. Account addresses are 20 bytes (160 bits) long and are represented in hexadecimal format.

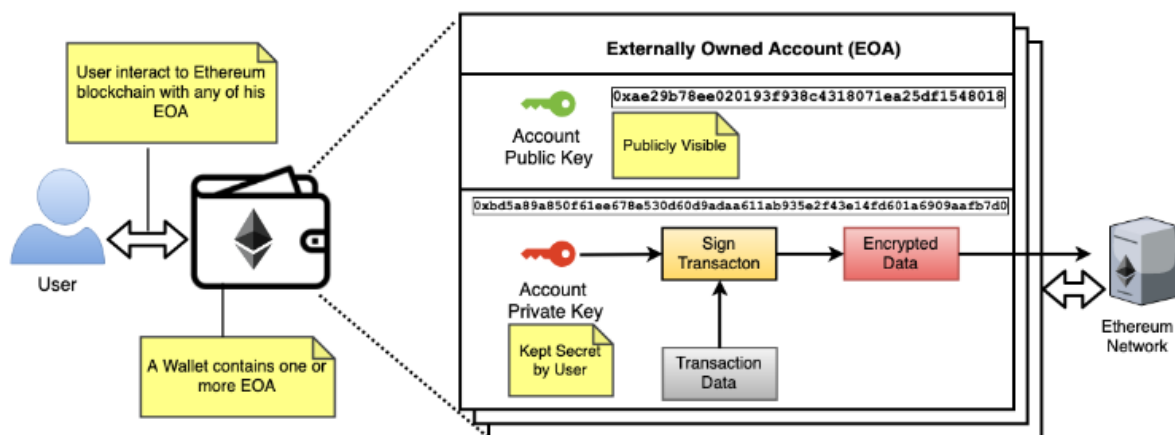
0x3fF34b4000308E581DC5b3CF009ad42b04479bAC

Externally owned accounts:

A wallet like Metamask holds your private key and from that private key it can derive a public key from which it can generate your account addresses. The private key needs to be kept secret, because that is what is required to sign transactions and transfer funds.

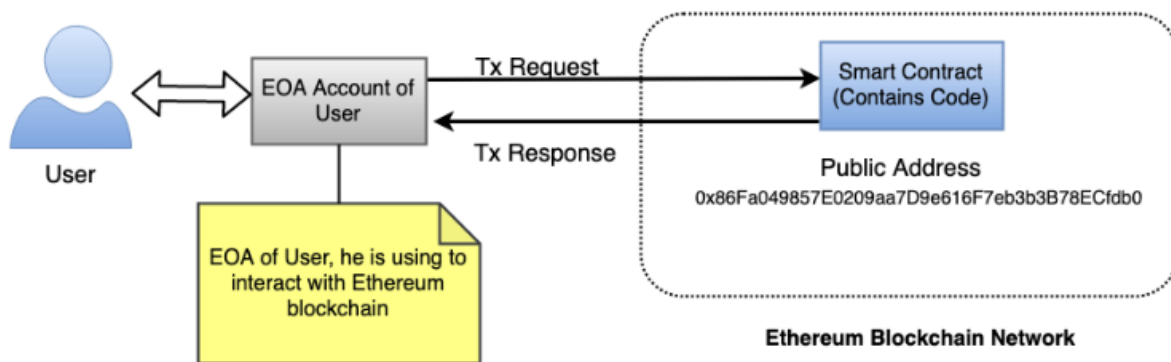
So, anyone who has your private key gets access to your funds. In order to backup and restore your private key, you will keep a 12 or 24 word key phrase (Mnemonic) somewhere safe, ideally on paper. On the other hand, your account address is public and can be shared with anyone. If someone wants to send you money, he will need your account address.

Your account addresses that are generated by Metamask from your private key can be used on all Ethereum networks as well as on other EVM compatible networks.



Contract Accounts:

When we deploy a smart contract from an EOA or from an already deployed contract) a similar 20 byte smart contract address is generated. This type of account can also hold an ETH balance and it stores the byte code of your smart contract. Contract accounts have no private key. A contract account is generated by sending a transaction without a value for the "to" field and the byte code in the "data" field (which is stored in the block with the transaction) from an EOA to the network.



Comparison between EOA and contract accounts:

	EOA	Contract account
Creation	A new account can be created at any time	A new contract can only be created from an EOA or from an existing deployed contract.
Public address	An EOA account's public address is derived from its private key.	A contract's public address is derived from the address of the creating account.
Control	Whoever owns the private key can control the funds of an EOA.	Control is defined by the code stored with it. An EOA or another contract may have control over a deployed contract.
Code	An EOA does not have any associated Solidity code.	A contract always has Solidity code associated with it, which is stored together with it.
Transaction initiation	Using the private key, one can directly access the funds of EOA and initiate any transactions.	A contract cannot initiate a transaction on its own. An EOA needs to initiate the transaction.

Transactions and Signatures

A transaction is required whenever we want to transfer money or change the state of a contract - in other words, whenever we want to modify something on the blockchain. A transaction can only be initiated from an EOA - for example from Metamask.

To create a transaction, we need to provide data for various fields and then we need to sign the transaction with our private key. That transaction is then sent to the Ethereum network, where it gets picked up by the miners and is included in a new block on the blockchain.

Fields of a transaction:

From: The address of the EOA that initiated the transaction.

Value: This indicates the amount of ETH that is sent to another EOA or a contract - this can be zero.

Max fee and max priority fee in Gwei: The fee you are willing to pay for the transaction.

Data: This represents the hexadecimal data that you want to transfer along with a transaction. This data field is required to call a method of a contract - the data field will hold the hex value of the encoded method and its arguments. When transferring ETH from one EOA to another EOA, the data field is empty,

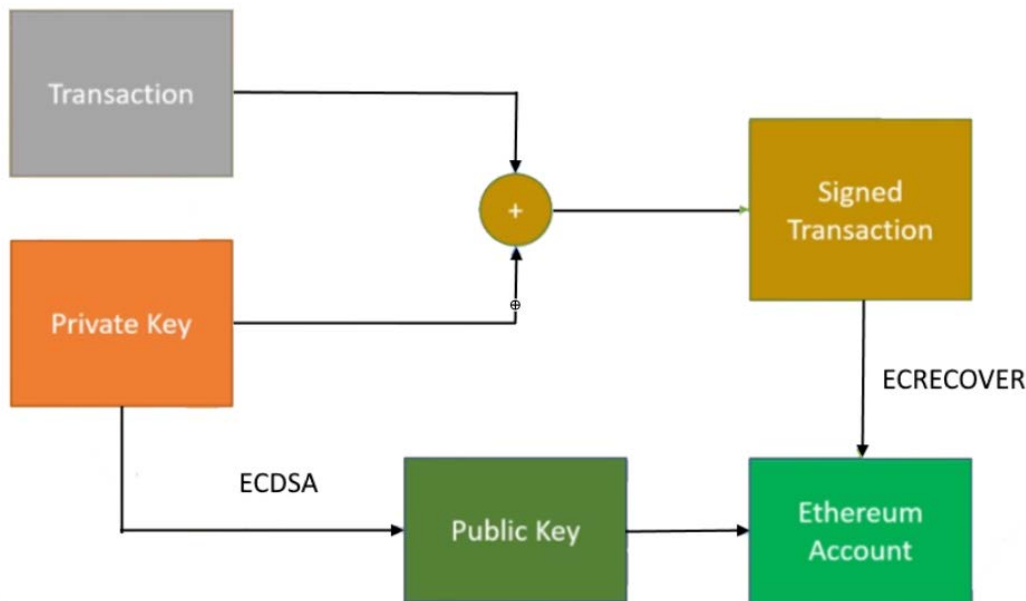
Example Transaction:

[illegible]

<https://web3js.readthedocs.io/en/v1.7.4/web3-eth.html>

The v, r and s fields are cryptographic data that are generated from the senders' private key and they can be used to re-generate the address of the senders account and therefore verify the validity of the transaction.

Creation and verification of a signed transaction:



Successfully mined transaction:

Overview	
Transaction Hash:	0x2ae6617a2abb6d8082d859cb3d8c4eef85ce81f91d373bdb63548fb64a0915e2
Status:	Success
Block:	15016946 1852 Block Confirmations
Timestamp:	8 hrs 14 mins ago (Jun-24-2022 06:29:59 AM +UTC) Confirmed within 30 secs
From:	0x57ebd61b13b32daf6c46cd069493208a017d72b
Interacted With (To):	Contract 0xb8c77482e45f1f44de1745f52c74426c631bdd52 (Binance: BNB Token)
Tokens Transferred:	From 0x57ebd61b13b32... To Binance U... For 0.629784 (\$278.91) BNB (BNB)
Value:	0 Ether (\$0.00)
Transaction Fee:	0.000845753095103994 Ether (\$1.01)
Gas Price:	0.000000023982110109 Ether (23.982110109 Gwei)

<https://etherscan.io/tx/0x2ae6617a2abb6d8082d859cb3d8c4eef85ce81f91d373bdb63548fb64a0915e2>

Ether, Wei and Gas

Ether or ETH is the native currency of the Ethereum blockchain. ETH can be subdivided into 18 decimal places and the smallest value is called Wei. 1 ETH contains 10^{18} Wei.

The most commonly used units are Wei, Gwei and ETH.

ETH conversion table:

Unit	Wei value	Wei	Ether value	Ether
Wei	1 wei	1	10^{-18} ether	0.000000000000000001
Kwei (KiloWei/babbage)	10^3 wei	1,000	10^{-15} ether	0.000000000000001
Mwei (MegaWei/lovelace)	10^6 wei	1,000,000	10^{-12} ether	0.000000000001
Gwei (GigaWei/shannon)	10^9 wei	1,000,000,000	10^{-9} ether	0.000000001
microether (szabo)	10^{12} wei	1,000,000,000,000	10^{-6} ether	0.000001
milliether (finney)	10^{15} wei	1,000,000,000,000,000	10^{-3} ether	0.001
ether	10^{18} wei	1,000,000,000,000,000,000	1 ether	1

Whenever we execute a transaction (in other words, whenever we want to modify anything on the blockchain) we need to pay a fee to execute that transaction. The amount of that fee depends on the complexity of the transaction - the number and nature of the code statements that need to be executed.

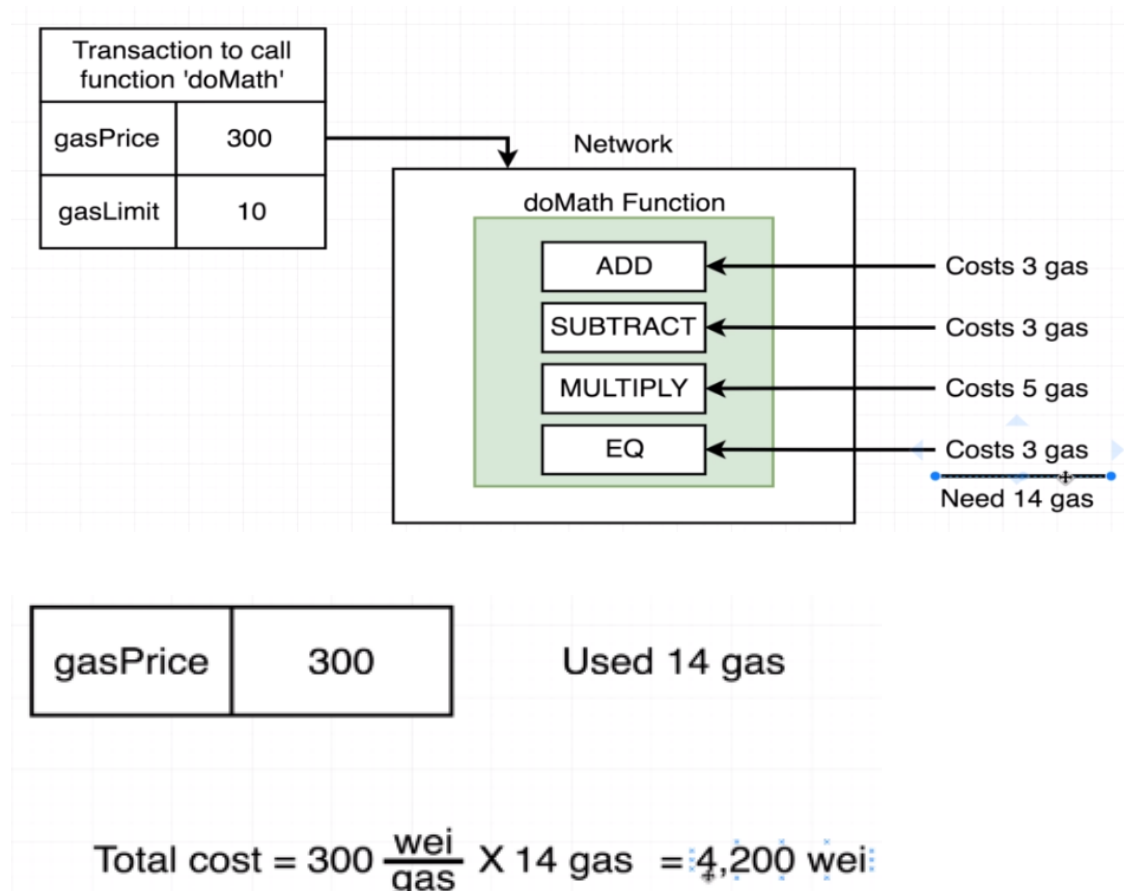
That fee is paid in units of gas. Gas is the fuel of the Ethereum blockchain network. Every operation (addition, subtraction...) consumes a certain amount of gas units and the price per unit of gas depends on the state of the network. If the network is congested, the price per unit of gas will be higher.

Gas limit: The gas limit is the maximum amount of gas units you are willing to pay for your transaction. If your transaction takes less gas, then the extra gas that you have provided will be refunded back to your wallet. If your transaction consumes all the gas and requires more gas to be executed, your transaction will fail.

Gas price: The gas price is the price per unit of gas you are willing to pay for executing your transaction. The gas price is always defined in Gwei. If you set a higher gas price, your transaction may be processed quicker.

EVM Opcodes: <https://ethereum.org/en/developers/docs/evm/opcodes/>

Calculation of transaction fees:



Example:

Transaction Fee:	0.000845753095103994 Ether (\$1.01)
Gas Price:	0.000000023982110109 Ether (23.982110109 Gwei)
Gas Limit & Usage by Txn:	70,532 35,266 (50%)
Gas Fees:	Base: 21.982110109 Gwei Max: 34 Gwei Max Priority: 2 Gwei
Burnt & Txn Savings Fees:	🔥 Burnt: 0.000775221095103994 Ether (\$0.93) 💰 Txn Savings: 0.000353290904896006 Ether (\$0.42)

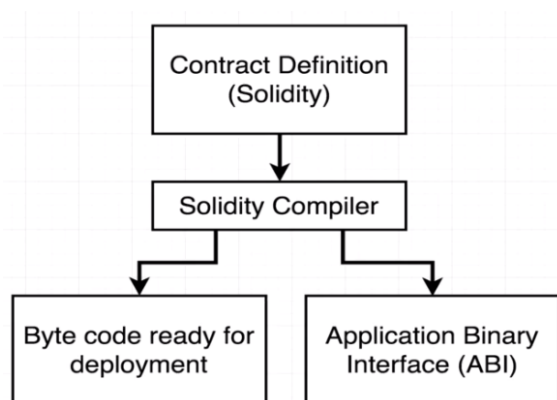
<https://etherscan.io/tx/0x2ae6617a2abb6d8082d859cb3d8c4eef85ce81f91d373bdb63548fb64a0915e2>

Smart Contracts and Solidity

A smart contract is a piece of code that is running on the blockchain. It can be viewed as a state machine. In order to change state, a transaction needs to be executed and mined.

Most smart contracts are written in Solidity. Solidity is a Turing-complete (which means, in theory any computation problem can be solved), strongly typed language that is similar to JavaScript.

After compiling a Solidity smart contract we get 2 important pieces of information: The smart contract byte code and the Application Binary Interface (ABI). The byte code is required for the deployment of the smart contract to the Ethereum blockchain - this bytecode is stored in the contract account. The ABI is required to access specific contract functions from a client application. The ABI contains the specification of all public contract functions including the types of the function arguments and the types of the return values.



The Solidity code as well as the ABI and the byte code can be consulted on etherscan.io:

When we destroy a SC (by calling `selfdestruct()`), the SC will be removed from state (it will no longer be available in any future blocks). But, of course, all past interaction with the SC and the initially deployed bytecode will always be available on the blockchain.


Transactions
Internal Txns
Erc20 Token Txns
Erc721 Token Txns
Contract
Events
Analytics
Info

Code
Read Contract
Write Contract
Read as Proxy NEW
Write as Proxy NEW

Contract Source Code Verified (Exact Match)

Contract Name: **FiatTokenProxy**
Optimization Enabled:

Compiler Version **v0.4.24+commit.e67f0147**
Other Settings:

 **Contract Source Code** (Solidity)

```

1  /**
2   *Submitted for verification at Etherscan.io on 2018-08-03
3   */
4
5   pragma solidity ^0.4.24;
6
7   // File: zos-lib/contracts/upgradeability/Proxy.sol
8
9   /**
10    * @title Proxy
11    * @dev Implements delegation of calls to other contracts, with proper
12    * forwarding of return values and bubbling of failures.
13    * It defines a fallback function that delegates all calls to the address
14    * returned by the abstract _implementation() internal function.
15    */
16    contract Proxy {
17        /**
18         * @dev Fallback function.
19         * Implemented entirely in `_fallback`.
20         */
21        function () payable external {
22            _fallback();
23        }
24
25        /**

```

There are 2 categories of smart contract functions:

Function that modify state:

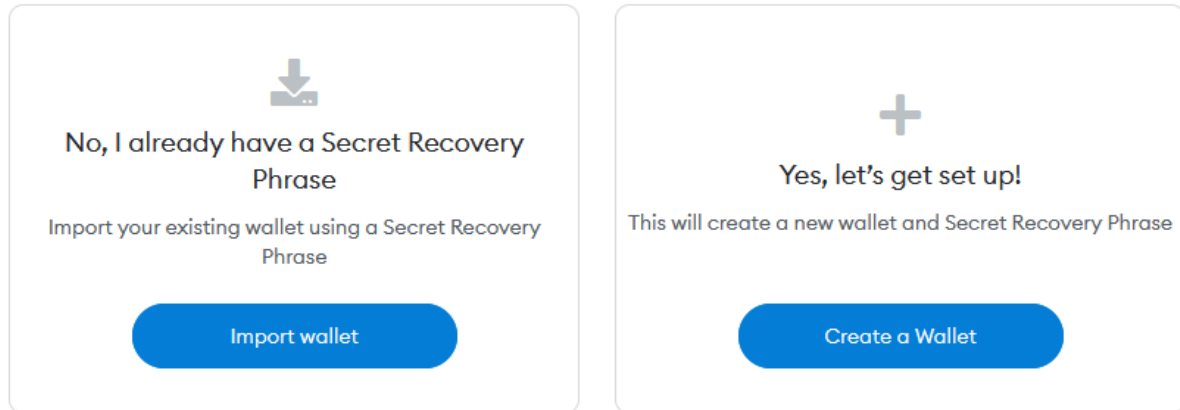
- A transaction is required to call the function
- Data on the blockchain is modified
- No data is returned - just a transaction hash
- The transaction first need to be mined before the function can run
- A transaction fee needs to be paid

Functions that return information:

- No transaction is required, a simple message call is sufficient
- No data on the blockchain is modified
- The function returns some data
- The function runs instantly
- Running the function does not cost anything

Setting up Metamask

Metamask plugin: <https://metamask.io/download/>




Make sure to save your secret recovery phrase - ideally on paper.

Now, you are automatically connected to the Ethereum Mainnet and you have one account

- Change the name of the account
- Copy the account address
- Create an additional account
- General settings: currency, language...
- Advanced settings: advanced gas controls, show test networks, customize transaction nonce, auto-lock timer
- Available networks: <https://chainlist.org/> => connect the app with Metamask

1 of 2

 https://chainlist.org


Connect With MetaMask

Select the account(s) to use on this site

Select all ⓘ


New Account

☒



Main Accou... (0xc7e...41f...
0 ETH

☐



Secondary ... (0xce2...25...
0 ETH

Only connect with sites you trust. [Learn more](#)






Cancel

Next

- Add the Polygon network

Networks

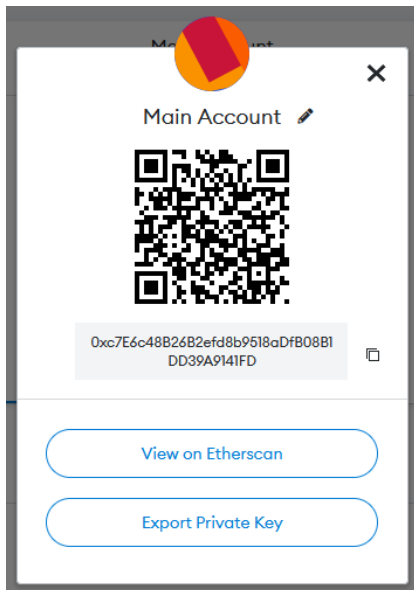
Add a network

<div><div></div> Ethereum Mainnet </div>	<div>Network Name</div> <div>Polygon Mainnet</div>
<div><div></div> Ropsten Test Network </div>	
<div><div></div> Rinkeby Test Network </div>	<div>New RPC URL</div> <div>https://polygon-rpc.com</div>
<div><div></div> Goerli Test Network </div>	<div>Chain ID ⓘ</div> <div>137</div>
<div><div></div> Kovan Test Network </div>	<div>Currency Symbol</div> <div>MATIC</div>
<div><div></div> Localhost 8545</div>	<div>Block Explorer URL (Optional)</div> <div>https://polygonscan.com</div>
<div><div></div> Polygon Mainnet</div>	

Cancel

Save

- Export private key



- Select the Ethereum main network and add a new token: USDC - <https://etherscan.io/>

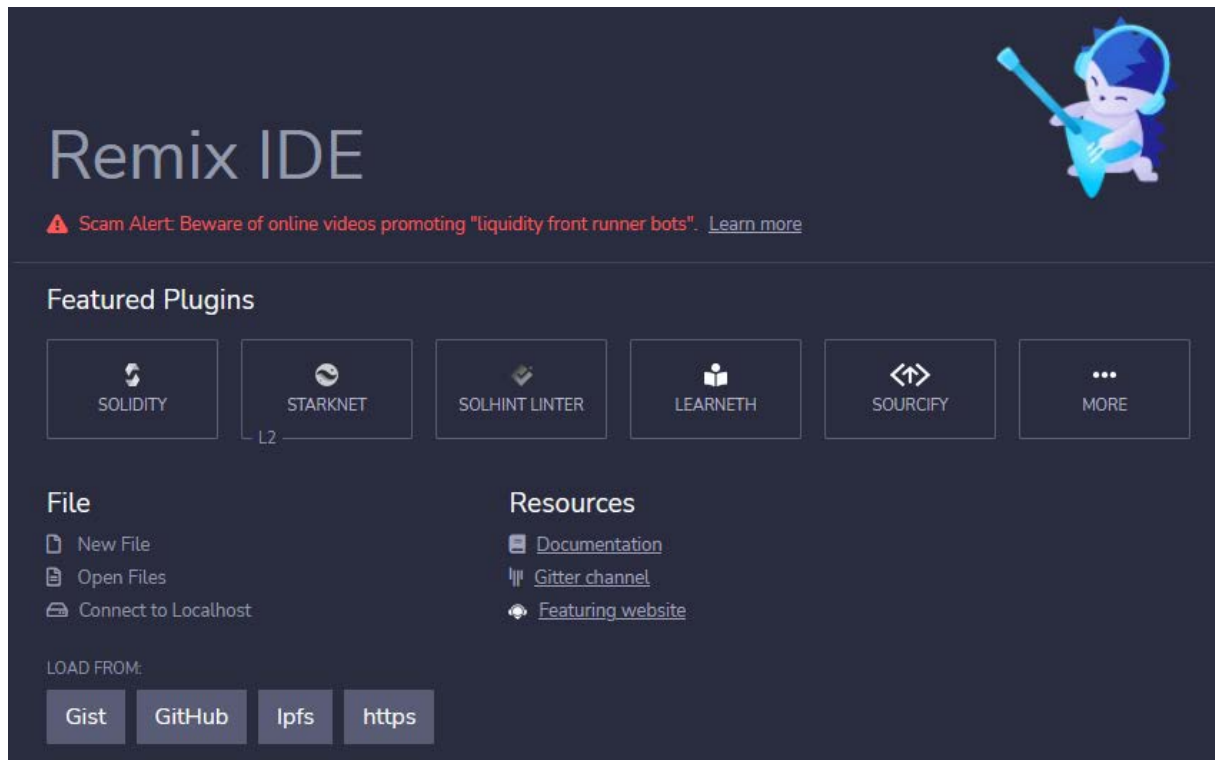
A screenshot of an 'Import Tokens' dialog box. It has a close button (X) in the top right. The title is 'Import Tokens'. Below the title is a section labeled 'Custom Token'. There's a warning box with an orange border and an information icon, containing the text: 'Anyone can create a token, including creating fake versions of existing tokens. Learn more about scams and security risks.' Below the warning are three input fields: 'Token Contract Address', 'Token Symbol', and 'Token Decimal'. The 'Token Decimal' field currently has the value '0' and a small up/down arrow icon on the right.

- Get Goerli Ether from faucet: <https://goerlifaucet.com/>, <https://goerli-faucet.pk910.de/>

Writing, compiling, deploying and debugging smart contracts on Remix

<https://remix.ethereum.org>

Home Page:

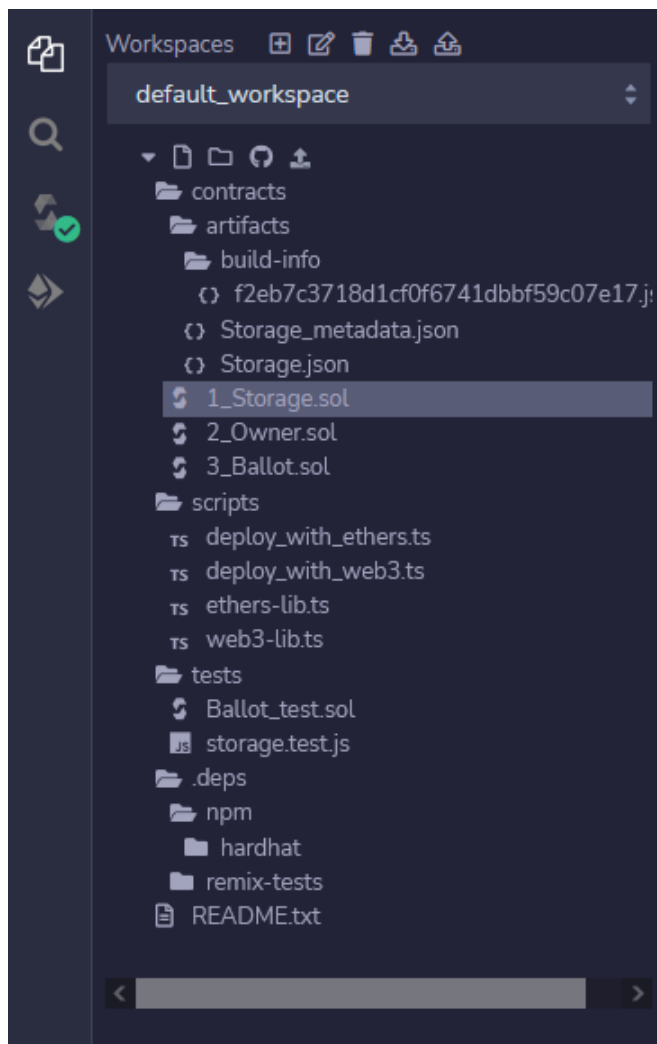


Possibility to add various plugins - the most important ones are already present. Many of those plugins are still in Alpha or Beta

We can also add files from Github (creates folder structure as on Github), IPFS or any other URL. Remidx allows a web socket connection between Remix and the local computer (practically Remix IDE makes available a folder shared by remidx)

However, there is no need to add files (eg: OpenZeppelin contract from Github) => we simply import the required file in our contract and the corresponding files are added automatically to the .deps folder: `import "@openzeppelin/contracts/token/ERC20/ERC20.sol";`

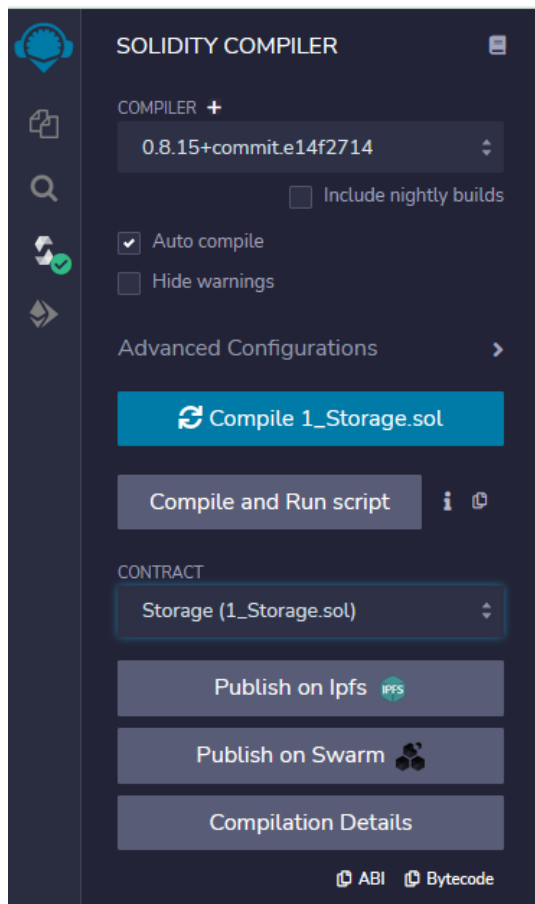
File Explorer:



When you launch the IDE, there is already default workspace with various sample contracts and scripts. You can also import and export your workspace. When you compile a contract, .json file with the ABI and byte code is created for you automatically - you could copy/paste the ABI to your PC and use it in a front-end project

We don't need the deployment scripts, because we can deploy our contracts directly from the Deploy & Run plugin.

Compiler:



Select the desired compiler version and "Auto compile"

IPFS: The Interplanetary File System (IPFS) is a decentralized file system. IPFS is a distributed system for storing and accessing files, websites, applications, and data. HTTP downloads files from one server at a time — but peer-to-peer IPFS retrieves pieces from multiple nodes at once, enabling substantial bandwidth savings. Today's web is centralized, IPFS is decentralized.

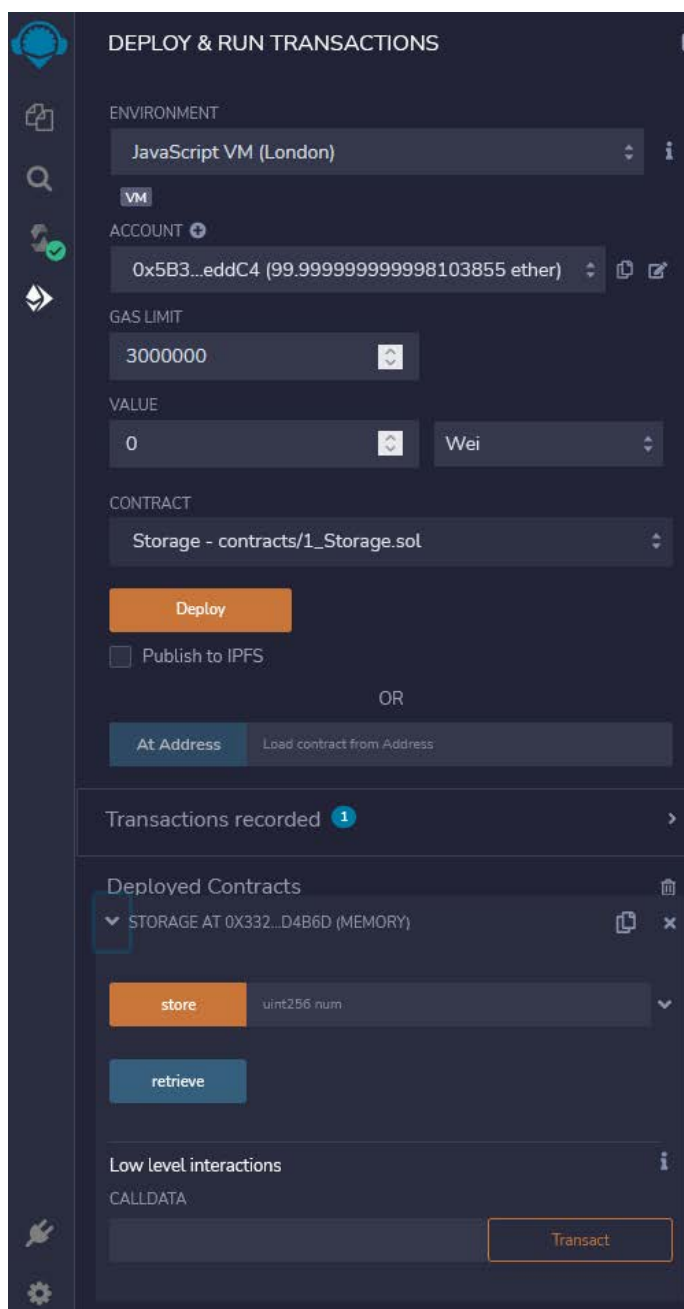
Swarm: Swarm is a system of peer-to-peer networked nodes that create a decentralized storage and communication service.

Deploy & Run Transactions:

These non-backward compatible hard forks can be seen as stepping stones for the Ethereum Network on the way towards "Serenity", or Ethereum 2.0. they've gotten their names from the cities in which the Ethereum "Devcon" developer conference took place.

Berlin HF: Ethereum's EIP-2565 proposal is about a change to how gas prices are defined

London HF: EIP-1559 "Fee Market Change for ETH 1.0 Chain". This will make Ethereum deflationary. According to EIP-1559's simple summary, EIP-1559 will contain a "transaction pricing mechanism" that involves a fixed network fee which is burned, and which dynamically increases or reduces block sizes in order to combat network congestion.



Environment:

JavaScript VM: Connects to an in-memory node managed by Remix => for testing during development: instantly deployed

Injected Web3: Uses the execution environment provided by Metamask - according to the selected network & account. => Connect Remix with Metamask (allow site) => eg: to test with a testnet once code is more robust : takes a few seconds to deploy (mining)

Web3 Provider: Connect to a local node - eg : Geth => http://127.0.0.1:8545

Hardhat, Ganache: Other in-memory nodes => hardhat integrates with Waffle testsuite, Ganache: nice desktop app that provides additional information about transactions, logs and events

Select account, leave gas limit as it is, if required, provide a value for your transaction (add a payable constructor - uint256 public balance;).

Demonstrate an error in the code => message in Compile tab. => show difference between read/write function (gas cost)

Hello World Smart Contract

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

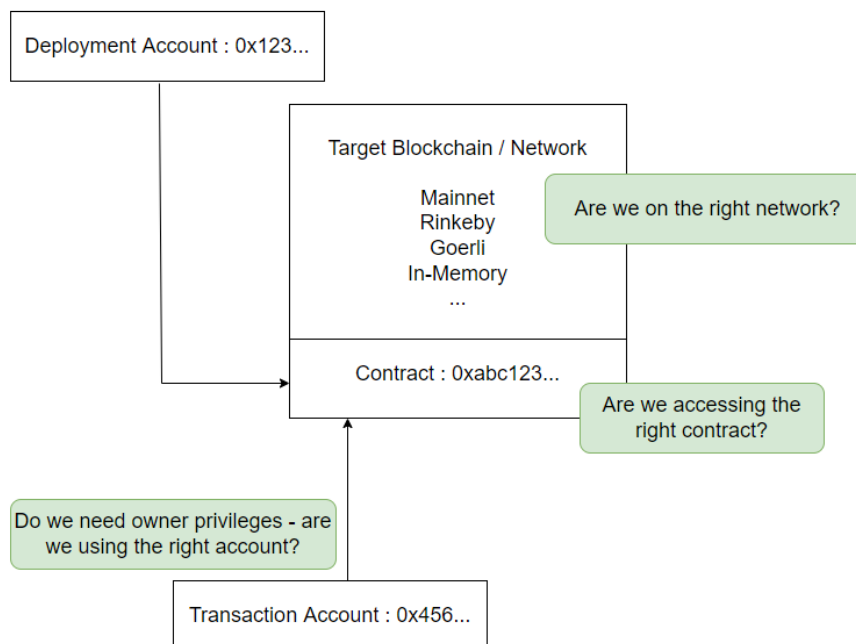
contract HelloWorld {
    string public message;

    event UpdateMessage(address indexed from, string oldStr, string newStr);

    constructor(string memory initMessage) {
        message = initMessage;
    }

    function updateMessage(string memory newMessage) public {
        string memory oldMsg = message;
        message = newMessage;
        emit UpdateMessage(msg.sender, oldMsg, newMessage);
    }
}
```

Deploy contract: we can also load an already deployed contract from a specific address



Setting up a local development environment

Install Node.js (includes NPM) - LTS version - Long Term Support:

<https://nodejs.org/en/>

As an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications. This is in contrast to today's more common concurrency model, in which OS threads are employed. Thread-based networking is relatively inefficient and very difficult to use. Furthermore, users of Node.js are free from worries of dead-locking the process, since there are no locks.

Install VS Code:

<https://code.visualstudio.com/download>

VS Code Plugins: Solidity + Hardhat, Live Server, Prettier, Emoji...

Notes: For Prettier, add a .prettierrc file to the top parent folder (not the project folder => add it to the Alchemy folder and not the eg: ERC20 folder)

Hardhat Plugin:

<https://marketplace.visualstudio.com/items?itemName=NomicFoundation.hardhat-solidity>

Code completion (msg.sender...) - format document (Prettier) - hover (shows function sig) - code validation (eg: msg.send instead of sender) - code fix (function without visibility - eg: public)

Compiling, testing and deploying smart contracts with Hardhat

What is Hardhat?

Hardhat is a development environment to compile, deploy, test, and debug Ethereum smart contracts. Hardhat comes built-in with the Hardhat Network (a local Ethereum network for development) and the Hardhat Runner - the CLI to interact with Hardhat that acts as an extensible task runner. To see all available tasks, run: `npx hardhat`

For example, running the compile task: `npx hardhat compile`

Requirements to run Hardhat:

Install the latest LTS version of Node.js: <https://nodejs.org/en/download>

Hardhat documentation: <https://hardhat.org/getting-started>

Creating a Hardhat project:

To setup a basic Hardhat project, execute the following commands:

- In your project folder, create a package.json file: ***npm init***
- Install Hardhat locally: ***npm i hardhat -D*** => this installs Hardhat as development dependency
- Create a hardhat project: ***npx hardhat*** => select: Create a basic sample project
- Install the following plugins: ***npm i -D @nomiclabs/hardhat-ethers ethers @nomiclabs/hardhat-waffle ethereum-waffle chai***

Configuring Hardhat:

The configuration file (hardhat.config.js) in the root folder of your project is always executed on startup before anything else happens - for example when we run a task.

Sample configuration file: <https://hardhat.org/config>

```
module.exports = {
  defaultNetwork: "rinkeby",
  networks: {
    hardhat: {
    },
    rinkeby: {
      url: "https://eth-rinkeby.alchemyapi.io/v2/123abc123abc123abc123abc123abcde",
      accounts: [privateKey1, privateKey2, ...]
    }
  },
  solidity: {
    version: "0.5.15",
    settings: {
      optimizer: {
        enabled: true,
        runs: 200
      }
    }
  },
  paths: {
    sources: "./contracts",
    tests: "./test",
    cache: "./cache",
    artifacts: "./artifacts"
  },
  mocha: {
    timeout: 40000
  }
}
```

It is not required to add hardhat or localhost (http://127.0.0.1:8545) to the list of networks - they are always available.

Compiling a Hardhat project

To compile all the smart contracts in your project, execute: `npx hardhat compile`

Compiling smart contracts with Hardhat generates two files per compiled contract: an artifact and a debug file. The artifact has all the information that is required to deploy and interact with the contract: contractName, abi, bytecode...

The HRE has an artifacts object with helper methods. For example, you can get a list with the paths to all artifacts by calling `hre.artifacts.getArtifactPaths()`. You can also read an artifact

using the name of the contract by calling `hre.artifacts.readArtifact("Bar")`, which will return the content of the artifact for the Bar contract.

Deploying smart contracts with Hardhat

Sample deployment script in the scripts folder: `deploy.js`

```
async function main() {
  const [deployer] = await ethers.getSigners();

  console.log("Deploying contracts with the account:", deployer.address);

  console.log("Account balance:", (await deployer.getBalance()).toString());

  const Token = await ethers.getContractFactory("Token");
  const token = await Token.deploy();

  console.log("Token address:", token.address);
}

main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

To run the script: ***`npx hardhat run scripts/deploy.js --network <network-name>`***

The Hardhat Network:

By default, Hardhat will always create an in-memory instance of the Hardhat Network whenever a task (this could also be a script or tests) is executed and all of Hardhat's plugins (ethers.js, Waffle...) will connect directly to this network's provider.

It's also possible to run the Hardhat Network in a standalone fashion so that external clients (like MetaMask or your DAPP) can connect to it.

Deploying a smart contract to a local Hardhat Network

- Open a command window and run: ***`npx hardhat node`*** => this starts a local Hardhat Network, that exposes a JSON-RPC interface to the Network through: `http://127.0.0.1:8545`

- In a second command window, run: ***npx hardhat run scripts/deploy.js --network localhost*** => localhost needs to be specified explicitly, deploy.js is your deployment script
- If you need MetaMask, select the "localhost" network in Metamask and make sure, the RPC URL is: <http://127.0.0.1:8545>, you may also need to set the Chain ID to: 31337

Debugging with Hardhat

When running your contracts and tests on the Hardhat Network you can print logging messages and contract variables using ***console.log(...)*** in your Solidity code. To use the logging feature, you have to import `hardhat/console.sol` in your contract code:

```
pragma solidity ^0.6.0;

import "hardhat/console.sol";

contract Token {
    //...
}
```

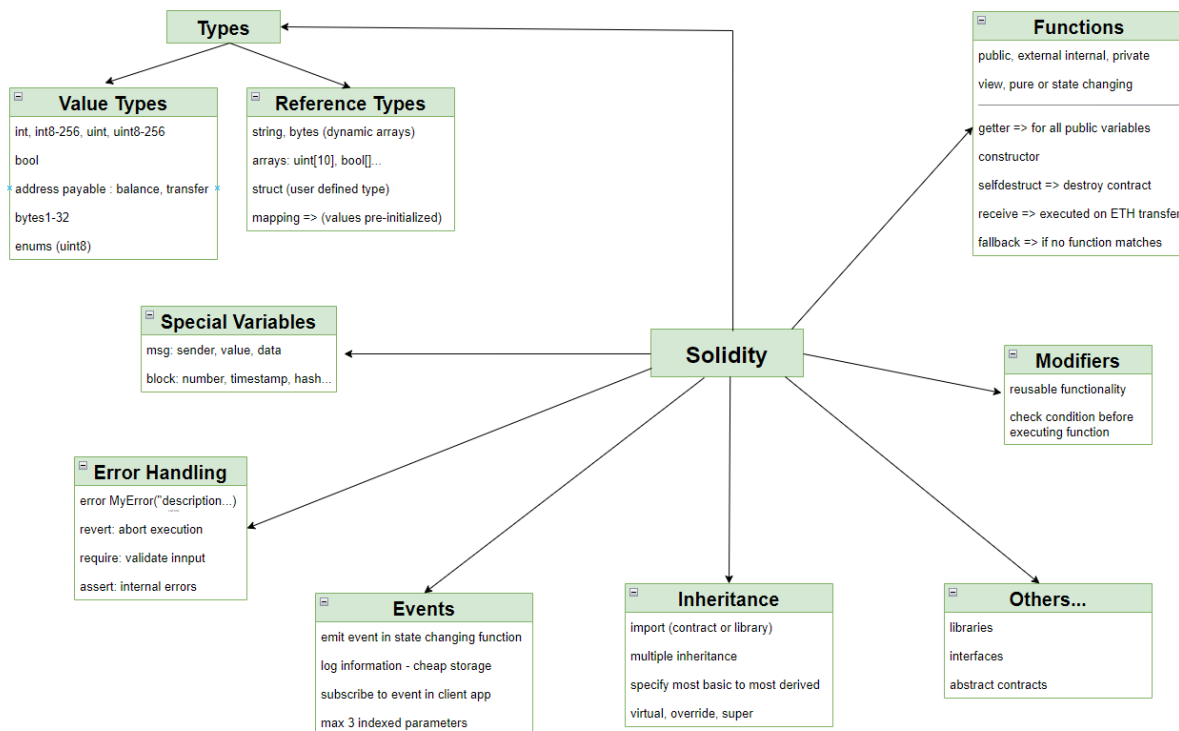
Now you can use logging in your contract functions:

```
function transfer(address to, uint256 amount) external {
    console.log("Sender balance is %s tokens", balances[msg.sender]);
    console.log("Trying to send %s tokens to %s", amount, to);

    require(balances[msg.sender] >= amount, "Not enough tokens");

    balances[msg.sender] -= amount;
    balances[to] += amount;
}
```

Solidity



Features and basic layout of a solidity smart contract:

- SPDX license identifier - the solidity compiler encourages the use of a SPDX License Identifier
- Pragmas - pragma solidity defines the compiler version to be used
- Imports
- Comments - standard comments and NatSpec comments
- State variables - permanently stored in contract storage
- Reference and value data types: int, uint, bool, address, arrays, string, bytes
- Mappings - initialized key/value pairs, like a hash table
- Struct - custom defined types that group several variables
- Enum - custom types with a finite set of values
- Events - to log specific information for client apps
- Modifiers - amend functions in a declarative way, for common checks applied to various functions.
- Errors - define descriptive names and data for failure situations: used in revert statements
- Constructor - executed once during creation of contract
- Functions
- Control structures (if / else) and loops (for / while)
- Inheritance - multiple inheritance is supported

```

// SPDX-License-Identifier: MIT
pragma solidity ^0.8.9;

import "../SomeOtherContract.sol";

contract Voting is SomeOtherContract {
    address public owner;

    struct Proposal {
        string name;
        uint voteCount;
    }

    mapping(address => Voter) public voters;

    Proposal[] public proposals;

    event UserHasVoted(address indexed voter, uint proposal);

    ///The user has already voted
    error UserHasAlreadyVoted(address voter);

    modifier onlyOwner() {
        require(msg.sender == owner, "You are not authorized");
        _;
    }

    constructor(string[] memory proposalNames) {
        owner = msg.sender;
        voters[owner].allowedToVote = true;
    }

    function giveRightToVote(address voter) external onlyOwner {
        if (voters[voter].voted) revert UserHasAlreadyVoted(voter);
        voters[voter].allowedToVote = true;
    }
}

```

Solidity Types

- Solidity is a statically typed language - the type of each variable needs to be specified
- All variables are initialized by default: `int = 0` ; `bool = false` ; `string = ""`...
- There is no null or undefined
- There are 2 categories of types: Value and Reference Types. When we use a reference type, we also need to specify the location of the variable: memory (lifetime limited to external function call), storage (area where state variables are stored - lifetime is limited to lifetime of the contract) or calldata (special data location for function arguments - behaves like memory)

Visibility of State variables

The value of state variables is stored permanently in contract storage.

- `public`: The compiler automatically generates a getter function with the name of the variable. Can be accessed internally via the name of the variable and externally via the getter function.
- `internal`: Can only be accessed from within the contract and from derived contracts. This is the default visibility
- `private`: Can only be accessed from within the contract

Value Types

- Value types: `int`, `uint`, `bool`, `address`, fixed size byte arrays (`bytes1`, `bytes2`...)
- Value types provide an independent copy of the value
- Fixed point numbers (`fixedMxN`, `ufixedMxN` - M bits and N decimal points) are not implemented yet by Solidity. They can be declared as custom type and operations like addition, subtraction... must be defined manually
- Workaround: define a variable that specifies the number of decimal places:
`uint numTokens = 10000;`
`uint decimalPlaces = 2;`
`=> this gives us 100,00 tokens (10000 / 10 ** 2)`
`=> for example: 10000 - 1099 = 8901 == 89,01`

Integers and unsigned integers

- Default value is 0
- `uint8` to `uint256` in 8 bit increments
- `uint8` from 0 to 255 ($2^{**} 8 = 256$)
- `uint256` from 0 to $(2^{**} 256) - 1$

- int8 from -128 to 127
- uint / int is an alias for uint256 / int256
- Exponentiation is only available for unsigned types via the ** parameter
- Solidity (since version 0.8.0) throws an error and reverts all state changes when we leave the defined value range (on overflow and underflow). To obtain the previous behavior (wrapping mode) an "unchecked {...}" block needs to be used.

```
uint public myInt = 0;

function decrement() public {
    myInt = 0;
    console.logUint(myInt);
    myInt--; //throws an error and reverts state
}
```

Boolean

- Default value is false

Address

- Default value is 0x0000...
- Every interaction on Ethereum is address based
- 20 byte value (40 characters)
- 2 Types: address and address payable
- Conversion from address to address payable must be explicit via payable(myAddress)
- A contract type can be converted to address or address payable: payable(address(this)) (if it can receive Ether - has a payable fallback or receive function)
- address member: balance
- address payable members: transfer (revert on failure) and send (returns false on failure => needs to be verified manually)
- send and transfer provide only 2300 gas => instead, use:
(bool success,) = receiverAddress.call{value: 1 ether}("");
require(success, "Transfer failed.");

```
constructor() payable {}

function transferEther() public {
    address payable addr1 = payable(0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2);
    address addrContract = address(this);
    if (addrContract.balance >= 10) addr1.transfer(10 ether);
}
```

Secure a smart contract with ownership:

```

address public owner;

constructor() {
    owner = msg.sender;
}

function sendMoney() public payable {

}

function getBalance() public view returns(uint) {
    return address(this).balance;
}

function withdrawAllMoney(address payable to) public {
    require(owner == msg.sender, "You are not allowed to withdraw money.");
    to.transfer(getBalance());
}

```

Transactions always need to be initiated from an EOA. If an EOA initiates a txn on a SC, that SC can initiate a txn on another SC

Fixed size byte arrays

- bytes1... bytes32: holds up to 32 bytes
- access individual bytes by index
- Members: length

Enums

- Can be used to create a user-defined type
- Stored as uint8 - if we have more than 256 values, than uint16
- Can be converted to and from integer types

```

enum Directions { Left, Right, Forward, Back }
Directions direction;

function setDirection(Directions dir) public {
    direction = dir;
}

function getDirection() public view returns (Directions) {
    return direction;
}

```

```
}
```

Reference Types

- Reference types: Structs, arrays (including bytes and strings) and mappings
- Reference types pass a reference and the value of the variable can be modified through different names.
- Required to specify the location of the variable:
 - memory: lifetime limited to external function call)
 - storage: area where state variables are stored, lifetime is limited to lifetime of the contract
 - calldata: special data location for function arguments, non-persistent, non-modifiable
behaves like memory
- Assignments between storage and memory always create an independent copy
- Assignments from memory to memory create references
- Assignments from storage to a local storage variable create a reference.
- All other assignments to storage always create a copy

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "hardhat/console.sol";

contract Test {
    uint[] arr1; //data location is storage - can be omitted

    function refTest(uint[] memory arr2) public {
        arr1 = arr2; //this creates an independent copy

        uint[] storage arr3 = arr1; //assigns a reference
        arr3[0] = 5; //modifies arr1 through arr3
        arr1[1] = 10; //modifies arr3 through arr1

        console.log(arr1[1]);
    }
}
```


Strings and Bytes:

- Dynamically sized special arrays - for bytes (raw data) or UTF-8 encoded strings
- String arrays don't have the length function and individual elements cannot be accessed by index
- Apart from `string.concat(s1, s2...)`, there are no string manipulation functions
- Using strings is expensive and should be avoided
- `bytes` is similar to `bytes1[]` but tightly packed (`bytes1[]` adds 31 padding bytes in memory between elements) - `bytes` is cheaper than `bytes1[]`
- If the length can be limited to a certain number of bytes, it is better to use one of the value types: `bytes1... bytes32`, because they are cheaper

Arrays:

- Can have compile-time fixed size (`uint[3] myArr`) or dynamic size (`uint[] myArr`)
- A getter is created for public arrays - only a single array element can be returned (the index needs to be specified on the getter function)
- Members: `length`
- Additional members for dynamic storage arrays: `push`, `push(value)` and `pop()`
- Memory arrays can be created with the `new` operator: `uint[] memory arr = new uint[](7)... arr[0] = 1...` - they cannot be resized (the `.push()` function is not available)
- Array literals can be used to assign values to a fixed size array: `uint8[3] memory arr = [1, 2, 3]`

Struct

- User defined types that group several variables
- Types in a struct are initialized with their default values

```
struct Payment {
    uint amount;
    uint timestamp;
}

function sendMoney() public payable returns (uint, uint) {
    Payment memory payment = Payment(msg.value, block.timestamp);

    return (payment.amount, payment.timestamp);
}
```

Mapping

- Mappings are like hash tables - each value is pre-initialized
- Usage: `mapping(keyType => valueType) myMapping`
- Example: `mapping(address => uint public balances;`
- Mappings are accessed like arrays, but there are no index-out-of-bounds exceptions and all possible key/value pairs are already initialized => any possible key can be accessed
- Use mappings rather than arrays, because they are cheaper
- The keyType cannot be a struct, mapping or array (bytes and string are allowed)
- Mappings always have storage as data location
- Mappings cannot be used as parameter or return value for public functions
- A getter is created for mappings that are used as public state variable - the keyType needs to be used as parameter for the getter
- Mappings and structs are often used together
- Mappings cannot be iterated

```
mapping(address => uint) public balances;

function setBalance(address myAddr, uint myBalance) public {
    balances[myAddr] = myBalance;
}
```

Special variables

There are some special variables/objects that are always available in the global namespace.

msg

- Provides data about the current message call
- `msg.sender`
- `msg.value`
- `msg.data`

block

- Provides general information about the blockchain
- `block.timestamp` - in seconds since unix epoch
- `block.hash(uint blocknumber)`
- `block.chainId`
- `block.basefee`
- `block.number`

```
struct Payment {
    uint amount;
```

```

    uint timestamp;
}

struct AccountDetails {
    uint totalBalance;
    uint numPayments;
    mapping(uint => Payment) payments;
}

mapping(address => AccountDetails) public accounts;

function getBalance() public view returns(uint) {
    return address(this).balance;
}

function sendMoney() public payable {
    accounts[msg.sender].totalBalance += msg.value;
    accounts[msg.sender].numPayments++;

    Payment memory payment = Payment(msg.value, block.timestamp);
    accounts[msg.sender].payments[accounts[msg.sender].numPayments] = payment;
}

function withdrawMoney(address payable to, uint amount) public {
    require(amount <= accounts[msg.sender].totalBalance, "not enough funds");
    accounts[msg.sender].totalBalance -= amount;
    to.transfer(amount);
}

```

Error Handling

- Transactions are atomic - they either fail or succeed as a whole
- All errors and exceptions revert the state of the contract to its initial values
- **require**(bool condition, [string memory message]) - typically used to validate input parameters, reverts if the condition is not met. Returns remaining gas
- **assert**(bool condition) - used for internal errors, if such an error occurs, the code is probably flawed and needs to be fixed. Reverts if the condition is not met. assert causes an error of type Panic with an error code (uint). Consumes all gas.

Assert is also triggered, if: division or modulo by zero, out of bounds index is accessed, convert a value too big to enum...

- **revert**(string memory reason) or **revert MyCustomError()** - aborts execution and reverts state changes. Using custom errors is much cheaper than providing information in a string - a detailed explanation of the error can be provided via NatSpec comments. Returns remaining gas

```

/// the provided amount for the purchase is incorrect
/// the product costs 1 ETH
/// @param provided The amount provided by the user
error WrongAmount(uint provided);

function buySomethingFor1ETH() public payable {
    if (msg.value != 1 ether)
        revert("Incorrect amount.");

    // Better use a custom error
    if (msg.value != 1 ether)
        revert WrongAmount(msg.value);

    // Alternative way to do it:
    require(msg.value != 1 ether, "Incorrect amount.");

    // Perform the purchase...
}

```

```

mapping(address => uint64) public received;

function receiveMoney() public payable {
    uint64 receivedAmount = uint64(msg.value);
    assert(msg.value == receivedAmount);
    received[msg.sender] += receivedAmount;
}

```

Functions

- Modify state => require a transaction - costs gas
- Read state => function call is performed (against a single node) - is free
- View functions only read from state, but don't modify state
- Pure functions neither read nor modify state
- View functions can call other view and pure functions
- Pure functions can only call other pure functions
- The names of return parameters can be omitted
- You can either explicitly assign return variables and then leave the function (without a return statement), or you can provide one or more return values directly with the return statement

Statements that are considered to modify state:

- Modifying a state variable
- Emitting an event
- Creating a contract

- Using selfdestruct
- Sending Ether
- Calling a function that is not marked pure or view

Statements that are considered to read state:

- All above
- Accessing address.balance
- Accessing any of the members of block or msg (except msg.data)

```
function calculate(uint a, uint b) public pure returns (uint sum, uint prod)
{
    return (a + b, a * b);
}

// assign the returned tuple
(uint addition, uint multiplication) = calculate(3, 5);
```

```
uint someValue = 5;

function funcWrite(uint val) public {
    someValue = val;
}

function funcView() public view returns(uint) {
    return someValue;
}

function funcPure(uint val) internal pure returns(uint) {
    return val * 2;
}
```

Function visibility

- public: can be called internally and externally
- private: can only be called from within the contract
- external: can be called externally and from other contracts. To call an external function from within the contract, use: this.myFunction()
- internal: can only be called from within the contract and from derived contracts
- internal function calls are cheaper and more efficient

Function overloading

A contract can have multiple functions with the same name but with different parameter types.

```
function func(uint value) public pure returns (uint out) {
    out = value;
}

function func(uint value, bool really) public pure returns (uint out) {
    if (really)
        out = value;
}
```

Special Functions

getter functions

- The compiler automatically creates a getter function for all public state variables
- A getter function for an array returns only one element of the array - the element corresponding to the provided index
- To call a getter function from another contract, use: someContract.myStateVar();

```
contract Called {
    uint public data = 42;
}

contract Caller {
    Called c = new Called();
    function func() public view returns (uint) {
        return c.data();
    }
}
```

constructor

- Called only once during deployment of the contract
- Arguments can be provided
- Initialize state variables - for example the owner of the contract
- Needs to be marked as payable if funds should be transferred during deployment

selfdestruct

- renders the contract unusable
- transaction history remains on the blockchain

- specify address to which the remaining funds will be sent

```
mapping(address => uint) public balanceReceived;

address payable public owner;

constructor() {
    owner = payable(msg.sender);
}

function destroySmartContract() public {
    require(msg.sender == owner, "You are not the owner");
    selfdestruct(owner);
}
```

receive

- A contract can have 1 receive function. The receive function cannot have arguments, cannot return anything and must have external visibility and be payable.
- receive() external payable {...}
- receive() is executed on plain Ether transfers (via .send() or .transfer())
- The receive function can only rely on 2300 gas - not many operations can be performed (for example: logging)

```
event Received(address, uint);
receive() external payable {
    emit Received(msg.sender, msg.value);
}
```

fallback

- A contract can have 1 fallback function that has external visibility
- fallback() external [payable] - or:
fallback (bytes calldata input) external [payable] returns (bytes memory output)
- The fallback function is executed on a call to the contract if none of the other functions match the given function signature
- The fallback function can receive data, but in order to also receive Ether it must be marked payable

- If receive() does not exist, but a payable fallback function exists, the fallback function will be called on a plain Ether transfer.
- If neither a receive Ether nor a payable fallback function is present, the contract cannot receive Ether through regular transactions and throws an exception.

```
uint x;
fallback() external payable { x = 1; }
```

Modifiers

- Modifiers are used to add reusable functionality to functions in a declarative manner
- Typically used to check specific conditions before executing the function
- Arguments can be provided
- Multiple modifiers can be applied to a function - separated by whitespace and executed in the order presented
- The symbol _ in the modifier is replaced with the function body

```
uint public biddingEnd; //set in constructor
error TooLate(uint time);

modifier onlyBefore(uint time) {
    if (block.timestamp >= time) revert TooLate(time);
    _;
}

function bid() external payable onlyBefore(biddingEnd)
{
    //do something...
}
```

Inheritance

- Multiple inheritance and polymorphism (function of the most derived contract is executed)
- A function in a lesser derived contract can be called by specifying the contract: ContractName.functionName()
- To call the function one level higher up in the inheritance hierarchy, use: super.functionName()
- Use "is" to derive from one or more contracts: contract C is A, B => B is the most derived contract
- A derived contract can access all non-private members
- Use the virtual keyword on a function to allow overriding
- Use the override keyword to override a virtual function from a derived contract

Importing Files

- Re-use existing smart contracts and libraries, organize code, make code easier to maintain
- `import "filename"` => all global members of the specified file are imported into the current global scope - this pollutes the global namespace and is not recommended
- `import * as someNamespace from "filename"` => all global members are accessible via the provided namespace
- `import {function1 as alias, function2} from "filename"` => import only required members - an alias can be provided

Events

- State changing functions cannot return values (they return a txn hash) - that's why we use events
- Events allow to log specific information - they are also used as cheap data storage (storing data on the blockchain is very expensive)
- Event arguments are stored in the transaction log - a special data structure on the blockchain
- Those logs are associated with the address of the contract
- Log data cannot be accessed from within the contract
- Client applications can subscribe and listen to events (read the log data) through the RPC interface of a provider and react accordingly
- Up to 3 parameters can be indexed and searched for later - they get added to a special data structure called topics

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract EventTest {

    mapping(address => uint) public balance;

    event TokensSent(address from, address indexed to, uint amount);

    constructor() {
        balance[msg.sender] = 100;
    }

    function sendToken(address to, uint amount) public {
        require(balance[msg.sender] >= amount, "Not enough tokens");
        assert(balance[to] + amount >= balance[to]);
        balance[msg.sender] -= amount;
        balance[to] += amount;
    }
}
```

```

    emit TokensSent(msg.sender, to, amount);
}
}

```

```

event Transfer(address indexed from, address indexed to, uint256 value);

function _transfer(address sender, address recipient, uint256 amount) internal
{
    // transfer tokens...
    emit Transfer(sender, recipient, amount);
}

```

```

logs
[
  {
    "from": "0x1c91347f2A44538ce624538EBd9Aa907C662b4bD",
    "topic": "0xe607861baff3d292b19188affe88c1a72bdc69d3015f18bb2cd0bf5349cc3e1",
    "event": "TokensSent",
    "args": {
      "0": "0x5B38Da6a701c56854dCfcB03FcB875f56beddC4",
      "1": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",
      "2": "5",
      "from": "0x5B38Da6a701c56854dCfcB03FcB875f56beddC4",
      "to": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",
      "amount": "5"
    }
  }
]

```

Deploy contract to Goerli:

Transaction Details
<
>

Overview
Logs (1)
State

Transaction Receipt Event Logs

62

Address
0x85c8664854416c212ae4b7189b2878eb7e6e3760

Topics

0

0x88a5966d370b9919b20f3e2c13ff65706f196a4e32cc2c12bf57088f88525874

1

→ 0x00000000000000000000000000000000b1b504848e1a5e90aeaa1d03a06ecee55562803

Data
Hex
→ 0016345785d8a0000

The "to" address is indexed and therefore listed as topic - together with topic0 for the entire event.

Libraries

- Defined by "library" keyword
- Provide re-usable set of functions - extend functionality of smart contract
- Cannot have state variables
- Cannot receive Ether
- Code is executed in the context of the calling contract - storage from calling contract can be accessed (only if state variables are explicitly supplied)
- First parameter of library function is often called "self" if the function can be seen as a method of that type - also, this parameter is of type storage reference
- The directive using libraryName for someType; attaches the library functions to someType. These functions will receive the object they are called on as their first parameter.

```
library Search {
    //call first parameter `self`, if function is method of that type.
    //first parameter is of type storage reference
    function indexOf(uint[] storage self, uint value)
    public view returns (uint)
    {
        for (uint i = 0; i < self.length; i++)
            if (self[i] == value) return i;
        return type(uint).max;
    }
}

contract Test {
    //attach all functions from the library "Search" to the type uint[]
    using Search for uint[];
    uint[] data;

    function replace(uint _old, uint _new) public {
        //the call to indexOf is compiled as delegatecall to external library
        uint index = data.indexOf(_old);
        //uint index = Search.indexOf(data,_old);
        if (index == type(uint).max)
            data.push(_new);
        else
            data[index] = _new;
    }
}
```

Abstract Contract and Interface

Abstract contracts

- Defined by "abstract" keyword
- A contract needs to be marked as abstract when one or more functions are not implemented
- A contract can be marked as abstract even if all functions are implemented
- An abstract contract cannot be instantiated directly
- Abstract contracts are used as a base class
- A contract inherits from an abstract contract by using the "is" keyword: `contract SomeContract is SomeAbstractContract`

Interfaces

- Defined by "interface" keyword
- Cannot have any functions implemented
- All functions must be external - they are virtual by default
- Cannot have state variables, constructor and modifier
- Can inherit from other interfaces
- A contract implements an interface by using the "is" keyword: `contract SomeContract is SomeInterface`

Voting smart contract

Requirements:

- During contract creation, all proposals are created.
- Each proposal has a name and holds the number of votes received.
- The contract has an owner (address that creates the contract) who can give other users the right to vote.
- Each user (who has the right to vote) can vote only once.
- After each vote, an event is emitted with the address of the voter and the voted proposal.
- At any time, anyone can check the currently leading proposal and its current number of votes.

Exercise:

- Add an endElection function that can only be called by the contract owner and after the election has ended
- Set the election end time in the constructor
- Make sure, the giveRightToVote function can only be called before the election end time - use a modifier that reverts with a custom error of type: ElectionHasEnded
- Revert with a custom error of type: EndElectionAlreadyCalled if the endElection function is called more than once
- Emit an ElectionEnded event when the endElection function is called and provide the name of the winning proposal and the vote count as event arguments

Using smart contracts from a web frontend - Ethers.js

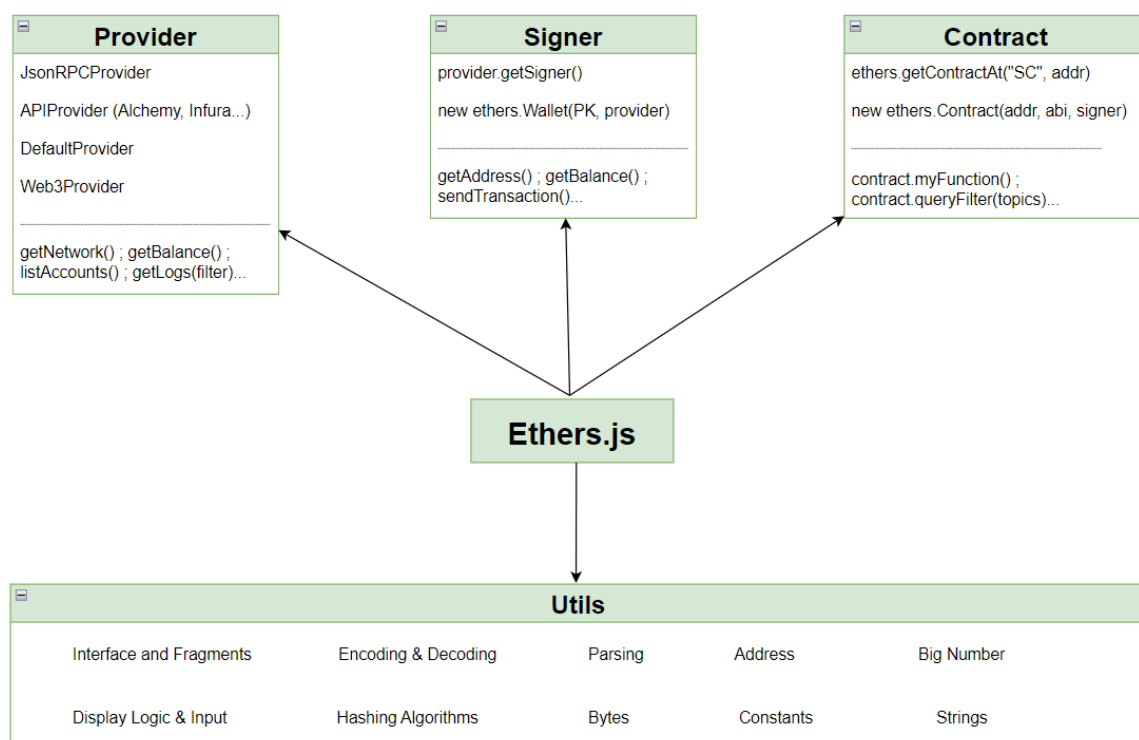
Ethers.js is a library for interacting with the Ethereum Blockchain. Ethers.js greatly facilitates the process of calling smart contract functions, but it also provides various utility classes that facilitate the interaction with the ABI, the treatment of Byte, String, Address and BigNumber types and much more.

The most important classes are: Provider, Signer and Contract.

The Provider is a class that provides a connection to the Ethereum Network - enables read-only access to the Blockchain.

The Signer is a class that has access to a private key (in order to authorize the network to charge your account ether to execute transactions) and can sign messages and transactions.

The Contract is a class that provides a connection to a specific contract on the Ethereum Network.



Provider

A Provider is an abstraction of a connection to the Ethereum network. A Provider in ethers is a read-only abstraction to access the blockchain data, a signer provides read/write access.

The ethers library offers default API keys for each service, so that each Provider works out-of-the-box. These API keys are provided for low-traffic projects and for early prototyping.

API Provider

There are providers for Etherscan, Infura, Alchemy...

```
//Network can be string or chainId, if no apiKey is provided, a shared API key will be used
new ethers.providers.AlchemyProvider( [ network = "homestead" , [ apiKey ] ] )
```

DefaultProvider

```
const provider = ethers.getDefaultProvider(network = "mainnet", {
  alchemy: YOUR_ALCHEMY_API_KEY
});
```

```
const provider = ethers.getDefaultProvider((network = "goerli"));
```

Properties & methods:

```
await provider.getBalance("0x123...");
await provider.getTransactionCount("0x123..."); //nonce for the next txn
```

```
await provider.getGasPrice() //estimation of gas price in wei
await provider.getFeeData() //estimation for maxFeePerGas, maxPriorityFeePerGas
```

```
provider.on( filter , listener ) //Add a listener to be triggered for each event.
provider.off( filter [ , listener ] ) //If no listener provided, remove all listeners
await provider.getLogs(filter) // Returns array of logs matching the filter
```

```
//listen to event
let filter = {
  fromBlock: 5,
  toBlock: "latest",
  address: "0x4AC22D164eeF548Bc12DE9eae09a8a9d76bDa30E",
  topics: [ethers.utils.id("UpdatedMess(address,string,string)"]],
}
```

```
console.log("Logs: ", await provider.getLogs(filter))

provider.on(filter, (log) => {
  console.log("Log: ", log)
})
```

Log example:

[illegible]

address: contract address

data: event data - values of various event args

topics: encoded event name with all args and all indexed props (eg: address)

Signers

A Signer is an abstraction of an Ethereum Account, which can be used to sign messages and transactions and send transactions. The available operations depend on the sub-class used.

The most commonly used Signer is a Wallet. The Wallet class can sign transactions and messages using a private key. If we are connected to local node using a JsonRpcProvider, we can also use a JsonRpcSigner: `signer = provider.getSigner()`

Properties & methods:

```
const wallet = new ethers.Wallet( privateKey [ , provider ] )
```

```
await wallet.getBalance();
await wallet.getAddress();
await wallet.getTransactionCount();
await wallet.estimateGas( transactionRequest )
```

```
tx = {
  to: "0x8ba1f109551b...",
```



```
    value: utils.parseEther("1.0")
  }
}
```

```
await wallet.signTransaction(tx)
await wallet.sendTransaction(tx)
```

```
let tx = {
  to: "0xC4446E49037c398DC7C01189569951951c7d7053",
  value: ethers.utils.parseEther("1.0"),
}
let res = await wallet.sendTransaction(tx)
console.log(res)
```

On a local node (Geth, Parity, Ganache...):

```
const jsonRpcSigner = jsonRpcProvider.getSigner()
```

Contract

A Contract class is used to communicate with the Contract on-chain. This class needs to know what methods are available and it gets this information from the ABI.

```
//passing a Provider, creates a Contract with read-only access (calls)
const contract = new ethers.Contract(contractAddress, abi, signerOrProvider);
```

Properties:

```
contract.address
contract.provider
contract.signer
```

Calling read-only methods:

```
const result = await contract.METHOD_NAME( ...args )
```

If the call reverts (or runs out of gas), an error will be thrown which will include:
error.address, error.args, error.transaction

Calling write methods:

```
const TransactionResponse = await contract.METHOD_NAME( ...args )
```

Events - Logs & Filtering:

Logs and filtering are used quite often in blockchain applications, since they allow for efficient queries of indexed data and provide lower-cost data storage when the data is not required to be accessed on-chain.

When a Contract creates a log, it can include up to 4 pieces of data to be indexed by. The indexed data is hashed and included in a Bloom Filter, which is a data structure that allows for efficient filtering.

```
//Transfer is the name of the event (takes 2 arguments)
filterFrom = contract.filters.Transfer(myAddress, null)

// Search for transfers *from* me
const logsFrom = await contract.queryFilter(filterFrom [ , fromBlock [ , toBlock ] )

// get args of event
logsFrom[0].args

// Listen to incoming events:
contract.on(filterFrom, (from, to, amount, event) => {
  // do something...
});
```

ContractFactory

A ContractFactory is an abstraction of a contract's bytecode and facilitates deploying a contract.

```
const contractFactory = new ethers.ContractFactory( abi , bytecode, signer )
const contract = await contractFactory.deploy( ...args ) // constructor args

contract.address
contract.deployTransaction
```

```
//in Hardhat we can use:
const contractFactory = await ethers.getContractFactory("myContractName")
const contract = await contractFactory.deploy()
await contract.deployed()
```

Types

Network: name, chainId...

FeeData: gasPrice, maxFeePerGas, maxPriorityFeePerGas...

Block: hash, number, timestamp...

TransactionRequest: describes a transaction that is to be sent to the network: to, from, nonce, data, value, gasLimit, gasPrice, maxFeePerGas, maxPriorityFeePerGas...

TransactionReceipt: response after the transaction has been mined (after wait() has been called on the TransactionResponse): contractAddress, gasUsed, logs, confirmations, status...

Utilities

An ABI is a collection of Fragments, where each fragment specifies a Function, Error, Event or Constructor.

Properties:

59

```
fragment.type ⇒ string //function, event or constructor
fragment.inputs ⇒ Array< ParamType >
```

Fragment types: FunctionFragment, ErrorFragment, EventFragment, ConstructorFragment

ethers.utils.FunctionFragment.from(objectOrString) => FunctionFragment

Interface

The Interface Class abstracts the encoding and decoding required to interact with contracts.

```
const interface = new ethers.utils.Interface( abi )
const interface = new ethers.utils.Interface (["function test(address from, uint amount)", ... ])
```

Properties:

```
interface.fragments ⇒ Array< Fragment >
interface.events ⇒ Array< EventFragment >
interface.functions ⇒ Array< FunctionFragment >
```

```
interface.getSigHash("balanceOf(address)"); // '0x70a08231'
```

Encoding Data

```
//Returns the encoded data, which can be used as the data for a transaction
interface.encodeFunctionData( fragment [ , values ] ) => string< DataHexString >
```

```
interface.encodeFunctionData("transferFrom", [
  "0x8ba1f109551bD432803012645Ac136ddd64DBA72",
  parseEther("1.0")
])
```

Decoding Data

```
// Decoding tx.data get provided values for debugging
const txData = "0x23b872dd00000000000000000000000008ba1...";
interface.decodeFunctionData("transferFrom", txData);
```

```
//Returns the decoded values from the result of a call for fragment
resultData = "0x0000000000000000de0b6b3a7640000...";
iface.decodeFunctionResult("balanceOf", resultData)
```

Parsing

Search the ABI for a matching Events, Errors or Functions and decode the components (name, arg, signature...)


```

Parsed log: LogDescription {
  eventFragment: {
    name: 'UpdatedMessages',
    anonymous: false,
    inputs: [ [ParamType], [ParamType], [ParamType] ],
    type: 'event',
    _isFragment: true,
    constructor: [Function: EventFragment] {
      from: [Function (anonymous)],
      fromObject: [Function (anonymous)],
      fromString: [Function (anonymous)],
      isEventFragment: [Function (anonymous)]
    },
    format: [Function (anonymous)]
  },
  name: 'UpdatedMessages',
  signature: 'UpdatedMessages(address,string,string)',
  topic: '0x8e47a29f32aeccd7c983742f80f48a82fe4cdf52fa10c202275d293555',
  args: [
    '0x7fA7776cdC16DC5B9752191d30087d383DF8c61a',
    'this is the new message',
    'test message',
    from: '0x7fA7776cdC16DC5B9752191d30087d383DF8c61a',
    oldStr: 'this is the new message',
    newStr: 'test message'
  ]
}

```

Addresses

```

//Injects the checksum (via upper-casing specific letters)
//throws if a checksummed address is provided, but a letter is the wrong case
ethers.utils.getAddress("0x8ba1f109551bd432803012645ac136ddd64dba72")
ethers.utils.isAddress( address ) ⇒ boolean
ethers.utils.computeAddress( publicOrPrivateKey ) ⇒ string< Address >

```

BigNumber

Many operations in Ethereum operate on numbers which are outside the range of safe values to use in JavaScript. A BigNumber is an object which safely allows mathematical operations on numbers of any magnitude. Most operations which need to return a value will return a BigNumber

```

//aBigNumberish: can be string, BigNumber, BigInt (JS), number (safe JS), BytesLike
ethers.BigNumber.from( aBigNumberish ) ⇒ BigNumber

```

```

BigNumber.from("42")
BigNumber.from("0x2a")
BigNumber.from(42)

```

Math operations:

`BigNumber.add(otherValue) ⇒ BigNumber`
sub, mul, div, mod, pow, abs

Comparison:

`BigNumber.eq(otherValue) ⇒ boolean`
lt, lte, gt, gte

Conversion:

`BigNumber.toBigInt() ⇒ bigint`
toNumber, toString, toHexString

Byte Manipulation

```
//Converts DataHexString to a Uint8Array
ethers.utils.arrayify("0x1234") // Uint8Array [ 18, 52 ]
```

```
//Converts a number or array to a HexString
ethers.utils.hexlify(1) // '0x01'
ethers.utils.hexlify([ 1, 2 ]) // '0x0102'
```

Constants

```
ethers.constants.AddressZero ⇒ string< Address > // 20 bytes
ethers.constants.Zero ⇒ BigNumber
ethers.constants.One ⇒ BigNumber
ethers.constants.WeiPerEther ⇒ BigNumber
ethers.constants.MaxUint256 ⇒ BigNumber
```

Display Logic and Input

A Dapp may specify the balance in ether, and gas prices in gwei for the User Interface, but when sending a transaction, both must be specified in wei. The `parseUnits` will parse a string representing ether, such as "1.1" into a `BigNumber` in wei. The `formatUnits` will format a `BigNumberish` into a string, which is useful when displaying a balance.

```
//Returns a string of value formatted with unit digits or to the unit specified
ethers.utils.formatUnits( value [ , unit = "ether" ] ) ⇒ string (in wei)
```

```
const oneGwei = BigNumber.from("10000000000");
formatUnits(oneGwei, 0); // '10000000000'
formatUnits(oneGwei, "gwei"); // '1.0'
```

```
formatUnits(oneGwei, 9); // '1.0'
```

```
ethers.utils.parseUnits( value [ , unit = "ether" ] ) ⇒ BigNumber (in wei)  
parseUnits("121.0", "gwei"); // { BigNumber: "121000000000" }  
parseUnits("121.0", 9); // { BigNumber: "121000000000" }
```

Hashing Algorithms

Create a hash from a string or bytes

```
//The Ethereum Identity function computes the KECCAK256 hash of the text  
//for example, to get the topic-hash of an event => returns a 32 byte DataHexString  
let topic = ethers.utils.id("UpdatedMessages(address,string,string)")
```

```
//Returns the keccak256 of bytes  
ethers.utils.keccak256( aBytesLike ) ⇒ string< DataHexString< 32 > >
```

```
ethers.utils.keccak256([ 0x12, 0x34 ]) // '0x56570de...'  
ethers.utils.keccak256("0x1234") // '0x56570de...'
```

```
// If needed, convert strings to bytes first:  
ethers.utils.keccak256(ethers.utils.toUtf8Bytes("hello world"))
```

```
// Or equivalently use the identity function:  
ethers.utils.id("hello world")
```

Solidity Hashing Algorithms

When using the Solidity `abi.encodePacked(...)` function, a non-standard tightly packed version of encoding is used. These functions implement the tightly packing algorithm.

```
// Returns the KECCAK256 of the non-standard encoded values  
//packed according to their respective type  
ethers.utils.solidityKeccak256( types , values ) ⇒ string< DataHexString< 32 > >  
ethers.utils.solidityKeccak256([ "int16", "uint48" ], [ -1, 12 ])
```

Strings

```
//Returns a bytes32 string representation of text.  
ethers.utils.formatBytes32String( text ) ⇒ string< DataHexString< 32 > >
```

```
//Returns the decoded string represented by the Bytes32 encoded data.  
ethers.utils.parseBytes32String( aBytesLike ) ⇒ string
```

```
ethers.utils.toUtf8Bytes( text ) ⇒ Uint8Array  
ethers.utils.toUtf8String( aBytesLike ) ⇒ string
```


Transactions

A generic object to represent a transaction.

Properties:

hash, to, from, nonce, data, value, gasLimit , maxFeePerGas, maxPriorityFeePerGas, v, r, s

```
//Parses the transaction properties from a serialized transaction.
```

```
ethers.utils.parseTransaction( aBytesLike ) ⇒ Transaction
```

```
ethers.utils.serializeTransaction( tx [ , signature ] ) ⇒ string< DataHexString >
```

```
tx = {  
  to: "0x4AC22D164eeF548Bc12DE9eae09a8a9d76bDa30E",  
  value: ethers.utils.parseEther("1.0"),  
  gasPrice: ethers.utils.parseUnits("50", "gwei"),  
  nonce: 10,  
  data: encodedFunction,  
}
```

```
let txSerialized = ethers.utils.serializeTransaction(tx)  
console.log("Tx Serialized: ", txSerialized)
```

```
//Parses the transaction properties from a serialized transaction.
```

```
txParsed = ethers.utils.parseTransaction(txSerialized)
```

```
console.log("Tx Parsed: ", txParsed)
```

The Metamask RPC API

MetaMask allows users to manage accounts and their keys. Whenever you request a transaction signature, MetaMask will prompt the user. MetaMask comes pre-loaded with fast connections to the Ethereum blockchain and several test networks via Infura. This allows you to get started without synchronizing a full node. MetaMask is compatible with any blockchain that exposes an Ethereum-compatible JSON RPC API (interface between client and full node)

For example, in order to retrieve the account balance, MetaMask, (or any other client) can make an HTTP request to an Ethereum node (full node), invoking the `eth_getBalance` RPC function with my address as a parameter. There are many providers that expose JSON-RPC interfaces, such as Alchemy, Infura...

Ethereum provider API

After installing MetaMask, your DAPP can interact with Metamask via the `window.ethereum` object. This API allows websites to request user accounts, read data from the blockchains the user is connected to, suggest that the user signs a transactions...

For any Ethereum web application you will have to:

- Detect the Ethereum provider (`window.ethereum`)
- Detect which Ethereum network the user is connected to
- Get the user's Ethereum account(s)

Important methods, events and requests:

To submit RPC requests to Ethereum via MetaMask we use: ***`ethereum.request(object)`***

Ethereum RPC API: <https://ethereum.org/en/developers/docs/apis/json-rpc/#json-rpc-methods>

Get the current network: `ethereum.networkVersion`

Get the selected address: `ethereum.selectedAddress`

Enable a DAPP on Metamask (select account(s) in popup):

```
const accounts = await ethereum.request({ method: 'eth_requestAccounts' });
```

Sending a transaction:

```
const transactionParameters = {  
  gasPrice: '0x09184e72a000', // customizable by user during MetaMask confirmation.
```

```

    to: '0x123...', // Required except during contract publications.
    from: ethereum.selectedAddress, // must match user's active address.
    value: '0x00', // Only required to send ether
    data: '0x7f746573743200000000000000000000...',
    chainId: '0x3', // Used to prevent transaction reuse across blockchains. Auto-filled by
    MetaMask.
  };

const txHash = await ethereum.request({
  method: 'eth_sendTransaction',
  params: [transactionParameters],
});

```

React if the user changes the account or the blockchain:

```

ethereum.on('accountsChanged', function (accounts) {
  // Time to reload your interface with accounts[0]!
});

ethereum.on('chainChanged', function (chainId) {
  // We recommend reloading the page
  window.location.reload();
});

```

Remove listeners once you are done listening to them:

```

ethereum.on('accountsChanged', handleAccountsChanged);
ethereum.removeListener('accountsChanged', handleAccountsChanged);

```

Errors

All errors thrown or returned by the MetaMask provider follow this interface:

```

interface ProviderRpcError extends Error {
  message: string;
  code: number;
  data?: unknown;
}

```

React DAPP Project

Requirements:

- Create a simple React web3 application: `npx create-react-app app-name`
- Retrieve accounts from Metamask and allow the app to be connected to one or more accounts: `request({method: "eth_requestAccounts"...`
- Whenever the app starts, verify if there are already any connected accounts: `request({method: "eth_accounts"...`
- Retrieve the current message value and display it in the UI
- Allow the message to be updated from the UI by sending a transaction to Metamask: `request({method: "eth_sendTransaction"...`
- Update the UI when the transaction has been mined and the message value has been updated on the blockchain: `contract.on("UpdateMessage"`

Hello World - React Metamask

Connected: 0xf39f...2266

Current Message:

test message

New Message:

Update the message in your smart contract.

🦊 Your message has been updated!

Update

Exercise:

interact.js:

- Add the provider
- Export the contract
- Return the initial message from the contract in loadCurrentMessage()
- Get all addresses from metamask => eth_requestAccounts in connectWallet()
- Get already connected Metamask accounts => eth_accounts and log a message whenever the network or an account changes => accountsChanged & chainChanged in getCurrentWalletConnected()
- In updateMessage() : encode the function we want to call ; set up transaction parameters => transactionParameters : to, from, data ; send the transaction using Metamask => eth_sendTransaction , return txHash

HelloWorld.js

- In useEffect() : load current message ; verify if a Metamask address is already connected with the app and retrieve the address and status
- In addSmartContractListener() : listen to the UpdateMessage event => setMessage, setNewMessage & setStatus
- In connectWalletPressed() : connect to Metamask and update the states
- In onUpdatePressed() : call updateMessage and update the state

ERC20 Tokens and OpenZeppelin

ERC20 Tokens

A Tokens can represent virtually anything in Ethereum: reputation points, skills of a character in a game, financial assets, a fiat currency like USD, an ounce of gold, real estate...

ERC20 (Ethereum Request for Comment) is the technical standard for fungible tokens on the Ethereum blockchain. A fungible token is one that is interchangeable with another token. This makes ERC20 tokens useful for things like a medium of exchange currency, voting rights, staking, and more. An ERC20 tokens is created by deploying a smart contracts.

Examples for ERC-20 Tokens: USDC, USDT, BNB, DAI, MAKER

ERC-20 contains several functions and events that a token must implement.

- `totalSupply()` → `uint256`: The total number of tokens that will ever be issued
- `balanceOf(address account)` → `uint256`: The account balance of a token owner's account
- `transfer(address recipient, uint256 amount)` → `bool`: Transfers a specified number of tokens to a specified address
- `transferFrom(address sender, address recipient, uint256 amount)` → `bool`: Transfers a specified number of tokens from a specified address to the specified recipient
- `approve(address spender, uint256 amount)` → `bool`: Allows a spender to withdraw a specified number of tokens from the caller's account
- `allowance(address owner, address spender)` → `uint256`: Returns the remaining number of tokens that spender will be allowed to spend on behalf of owner
- `Transfer(from, to, value)` : An event triggered when a transfer is successful
- `Approval(owner, spender, value)` : An event triggered when the allowance of a spender for an owner is set by a call to approve - value is the new allowance.

OpenZeppelin

A library of modular, reusable, secure smart contracts for the Ethereum network. OpenZeppelin Contracts help you minimize risk by using battle-tested libraries of smart contracts for Ethereum.

Provided contracts:

- Tokens: ERC20, ERC721, ERC1155...
- Access control: ownership, role-based access control
- Utilities: checking signatures, PaymentSplitter, Multicall...

Installation: `npm install @openzeppelin/contracts`

Usage:

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
```

```
contract MyToken is ERC20 { ...
```

ERC20 Token Project

Requirements:

- Create an ERC20 Token with the name: MyToken and the symbol: MT
- During deployment, mint 100.000 MT's to the contract deployer
- The token must provide the following properties, functions and events:
 - totalSupply()
 - balanceOf(address account)
 - transfer(address recipient, uint256 amount)
 - transferFrom(address sender, address recipient, uint256 amount)
 - approve(address spender, uint256 amount)
 - allowance(address owner, address spender)
 - Transfer(from, to, value) Event
 - Approval(owner, spender, value) Event
- Deploy the contract to the Goerli testnet
- Import the token into Metamask and send some tokens to a different address

Exercise:

Write a script that performs the following tasks on a local Hardhat node:

- Display the MT balance of the deployer account
- Transfer 1000 MT's from the deployer account to another Hardhat account
- Display the MT balance of the second account
- Allow the second account to spend up to 50 MT's from the deployer account
- Display how many tokens the second account can spend on behalf of the deployer account

Testing Smart Contracts

Tests are executed on the Hardhat network using ethers. Mocha is used as the test runner and Chai as assertion library. Additionally, custom Hardhat-Chai matchers and the Hardhat Network Helper plugins are used.

Basic structure of a test file

```
describe("Contract...", function () {
  async function myFixture() {
    ...
  }

  describe("Test group 1", function () {
    it("Some description...", async function () {
      const { var1, var2... } = await loadFixture(myFixture)
      ...
    });
    ...
    it("other test description...", async function () {
      ...
    });
  })
})

describe("Test group 2", function () {
  it("Some description...", async function () {
    ...
  });
  ...
})
});
```

Running the tests:

```
npx hardhat test test/my-tests.js --- or:
npx hardhat test --- or:
npx hardhat test --network hardhat
```

Hardhat Chai Matchers

This plugin adds Ethereum-specific capabilities to the Chai assertion library

Installation and usage:

```
npm install --save-dev @nomicfoundation/hardhat-chai-matchers
```

Add the following to `hardhat.config.js`:

```
require("@nomiclabs/hardhat-ethers");  
require("@nomicfoundation/hardhat-chai-matchers")
```

Numbers:

```
expect(await token1.totalSupply()).to.eq(1_000_000)  
expect(await token2.totalSupply()).to.eq(ethers.utils.parseEther("1"))  
expect(await token2.totalSupply()).to.eq(10n ** 18n)
```

Reverted transactions

reverted:

```
await expect(token.transfer(address, 0)).to.be.reverted;
```

revertedWith:

```
await expect(token.transfer(address, 0)).to.be.revertedWith(  
  "transfer value must be positive"  
);
```

revertedWithCustomError:

```
await expect(token.transfer(address, 0))  
  .to.be.revertedWithCustomError(token, "InvalidTransferValue")  
  .withArgs(0);
```

The first argument must be the contract that defines the error. If the error has arguments, the `.withArgs` matcher can be added

Events

```
await expect(token.transfer(address, 100))
  .to.emit(token, "Transfer")
  .withArgs(100);
```

Assert that a transaction emits a specific event. The first argument must be the contract that emits the event. If the event has arguments, the `.withArgs` matcher can be added.

Balance change

These matchers can be used to assert how a given transaction affects the ether balance, or an ERC20 token balance of a specific address.

changeEtherBalance:

```
await expect( wallet.sendTransaction({ to: receiver, value: 1000 }))
  .to.changeEtherBalance(wallet, -1000);
```

Assert that the ether balance of an address changed by a specific amount:

changeTokenBalance:

```
await expect(token.transfer(receiver, 1000))
  .to.changeTokenBalance(token, sender, -1000);
```

Assert that an ERC20 token balance of an address changed by a specific amount. The first argument must be the contract of the token.

Hardhat Network Helpers

This package provides convenience functions for quick and easy interaction with the Hardhat Network. Facilitates the manipulation of account attributes (balance, nonce...) and the ability to impersonate specific accounts.

Installation and usage:

```
npm install --save-dev @nomicfoundation/hardhat-network-helpers
```

```
const { setBalance , loadFixture } = require("@nomicfoundation/hardhat-network-helpers");
```

Set the balance for the given address:

```
await setBalance(address, 100);
```

Return the timestamp of the latest block:

```
await time.latest();
```

Mine a new block whose timestamp is newTimestamp:

```
await helpers.time.increaseTo(newTimestamp);
```

Take a snapshot of the state of the blockchain at the current block.

```
const snapshot = await takeSnapshot();
```

Returns an object with a restore method that can be used to reset the network to the state in the snapshot. After doing some changes, you can restore to the state of the snapshot

```
await snapshot.restore();
```

Fixtures:

In a typical Mocha test, the duplication of code (eg: deploying a contract) is handled with a `beforeEach` hook. However, if you have to deploy many contracts, your tests will be slower because each one has to send multiple transactions as part of its setup. The `loadFixture` helper in the Hardhat Network Helpers fixes this problem.

The first time `loadFixture` is called, the fixture is executed. But the second time, instead of executing the fixture again, `loadFixture` will reset the state of the network to the point where it was right after the fixture was executed. This is faster, and it undoes any state changes done by the previous test.

```
const { expect } = require("chai");
const { loadFixture } = require("@nomicfoundation/hardhat-network-helpers");
```

```
describe("Group...", function () {
  async function deployFixture() {
    const Contract = await ethers.getContractFactory("myContract");
    const contract = await Contract.deploy();

    return { contract, param1, param2... };
  }

  it("Description...", async function () {
    const { contract, param1, param2... } = await loadFixture(deployFixture);
    expect(await contract.someMethod()).to.equal(111);
  });
});
```

Requirements:

Add the following tests to the ERC20 project:

- Create a contract deployment fixture => return the contract and signers for the first 2 accounts
- Transfer 10 ETH to the second account and verify the token balance
- Call setBalance and verify the updated token balance
- Transfer to a zero address and verify if the call reverts with the message: ERC20: transfer to the zero address => to.be.revertedWith("...")
- Verify if the "transfer" event is emitted after a token transfer - verify also the event arguments => to.emit(...).withArgs(...)
- Verify if the token balance changes after a token transfer => changeTokenBalance

Exercise:

Add the following tests to the ERC20 project:

- Verify if the total token supply is correct
- Verify if the total token supply and the deployer token balance are equal after contract deployment

NFT's and ERC721 Tokens

What are NFT's

NFT stands for a non-fungible token, which means it is unique and not interchangeable because it has unique properties. NFTs are used to represent ownership of unique items - they can only have one official owner at a time. Each minted token has a unique identifier that is directly linked to one Ethereum address. An NFT is a digital (tokenized) asset that can represent all kinds of things, like: collectibles, game items, art, real estate...

The creator of an NFT can define the scarcity of the asset. The creator may create an NFT where only one is minted as a special rare collectible or where 1000 identical items are minted that may be used as admission tickets to an event (in this case, each NFT would still have a unique identifier - like a bar code on a traditional ticket - with only one owner.

ERC721 Tokens

<https://docs.openzeppelin.com/contracts/4.x/api/token/erc721>

ERC721 is a standard for non-fungible tokens. For example, ERC721 tokens can be used to track items in a game, which have their own unique attributes. Whenever one is awarded to a player, it will be minted and sent to the player's address.

IERC721:

- `balanceOf(owner)`
- `ownerOf(tokenId)`
- `transferFrom(from, to, tokenId)`
- `safeTransferFrom(from, to, tokenId)`
- `approve(to, tokenId)`
- `setApprovalForAll(operator, _approved)`
- `getApproved(tokenId)`
- `isApprovedForAll(owner, operator)`

Events

- `Transfer(from, to, tokenId)`
- `Approval(owner, approved, tokenId)`
- `ApprovalForAll(owner, operator, approved)`

`safeTransferFrom(address from, address to, uint256 tokenId)`

Safely transfers tokenId, checking that a recipient (if it is a smart contract) implements IERC721Receiver (IERC721Receiver.onERC721Received is called upon a safe transfer) to prevent tokens from being locked forever.

It must return its Solidity selector to confirm the token transfer. If any other value is returned or the interface is not implemented by the recipient, the transfer will be reverted.

The selector can be obtained with IERC721Receiver.onERC721Received.selector.

approve(address to, uint256 tokenId)

Gives permission to transfer tokenId to another account. The approval is cleared when the token is transferred. Only a single account can be approved at a time

setApprovalForAll(address operator, bool _approved)

Approve or remove token transfer to the specified address. Operators can call transferFrom or safeTransferFrom for any token owned by the caller.

getApproved(uint256 tokenId) → address operator

Returns the account approved for tokenId token.

isApprovedForAll(address owner, address operator) → bool

Returns if the operator is allowed to manage all of the assets of owner.

ERC721URIStorage

OpenZeppelin provides several specialized ERC721 contracts - one of them is the ERC721URIStorage contract that allows to provide different metadata (token properties) for each minted token.

```
tokenURI(tokenId)  
_setTokenURI(tokenId, _tokenURI)
```

ERC1155

ERC1155 is a multi-Token Standard - a single smart contract can represent multiple tokens at once. For example, the ERC721 balanceOf function refers to how many different tokens an account has, not how many of each.

This approach leads to massive gas savings for projects that require multiple tokens. Instead of deploying a new contract for each token type, a single ERC1155 token contract can hold the entire system state, reducing deployment costs and complexity.

This can be useful for example for a smart contract that needs to hold various items with specific properties (NFT's) for a game: Sword, shield, knife...

What is IPFS:

IPFS is a decentralized, peer-to-peer file-sharing system that allows to store and access files, websites, applications, and other data.

What is Pinata

Pinata is the leading media management company for web3 developers and it allows to PIN different types of data to IPFS.

When an IPFS node retrieves data from the network it keeps a local cache of that data for future usage, taking up space on that particular IPFS node. IPFS nodes frequently clear this cache out in order to make room for new content.

In order to prevent an IPFS node from deleting specific data, that data needs to be labeled specifically and that process is referred to as pinning (that's also where Pinata got its name from). So, pinning prevents important data from being deleted.

Pinata is an IPFS pinning service - it is a collection of IPFS nodes dedicated to pinning content on IPFS. When you pin your content to IPFS with Pinata your content will always be available online.

Advantages of using a pinning service:

- Speed of retrieving data
- Uptime of IPFS nodes
- Almost unlimited space

Pin JSON: <https://api.pinata.cloud/pinning/pinJSONToIPFS>

NFT Minter Project

Requirements:

ERC721 Token:


- Create an ERC721 token with the name: MyNFT and the symbol: MNFT
- The token should inherit from the ERC721URIStorage contract and only the contract owner should be able to mint NFT's => inherit from the ownable contract
- Implement a public mintNFT function that allows to associate a metadata file with the NFT and that manages the token Id (increment the Id each time a new NFT is minted and associate the Id with the provided tokenURI)
- The token must provide the following properties, functions and events:
 - balanceOf(account)
 - ownerOf(tokenId)
 - safeTransferFrom(from, to, tokenId)
 - transferFrom(from, to, tokenId)
 - approve(to, tokenId)
 - Transfer(from, to, tokenId) => Event
 - Approval(owner, approved, tokenId) => Event
- Deploy the contract to the Rinkeby testnet (that way we can display the NFT in the OpenSea testnet)
- Import the token into Metamask and send some tokens to a different address
- Upload an image file to IPFS via Pinata.
- Create a metadata file for the NFT with 2 properties, a name, a description and a link to the previously uploaded image file. Upload the metadata file to IPFS via Pinata
- Create a script file (scripts/nft-mint.js) that mints an NFT to a provided recipient address. Provide the link to the previously created metadata file in the mintNFT smart contract function


Web Application:


- Create a REACT web frontend with the following features:
 - A button to connect the DAPP to Metamask: request({method: "eth_requestAccounts"...
 - Provide various NFT metadata: name, description and a URL to an image file that is stored on IPFS
 - A button to mint an NFT with the provided metadata properties
 - Send the transaction data (minting the NFT) to Metamask: request({method: "eth_sendTransaction"...
- Use Pinata to pin the provided NFT metadata to a resource file that is stored on IPFS and that will be used as the tokenURI in the mintNFT smart contract function

NFT Minter


Connected: 0x46f9...c073

 Link to asset:
e.g. <https://gateway.pinata.cloud/ipfs/<hash>>

 Name:
e.g. My first NFT!

 Description:
e.g. Even cooler than cryptokitties ;)

Mint NFT

 Provide an image url, a name and a description for your NFT.

Exercise:

ERC721 Token:

Open the file `scripts/mint-nft.js` and provide the following features:

- Mint a NFT for a recipient that is not the token owner and provide a link to a metadata file that is hosted on IPFS
- Display the number of NFT's (of type MNFT) owned by the recipient
- Display the owner of the NFT with the Id of 1
- Transfer the NFT from the recipient to the contract owner

Web Application:

`interact.js`:

- In the `mintNFT` function, encode the function data for the "mintNFT" smart contract function
- Create the `txn` parameters (from, to and encoded function data)
- Send the transaction using Metamask => `eth_sendTransaction`

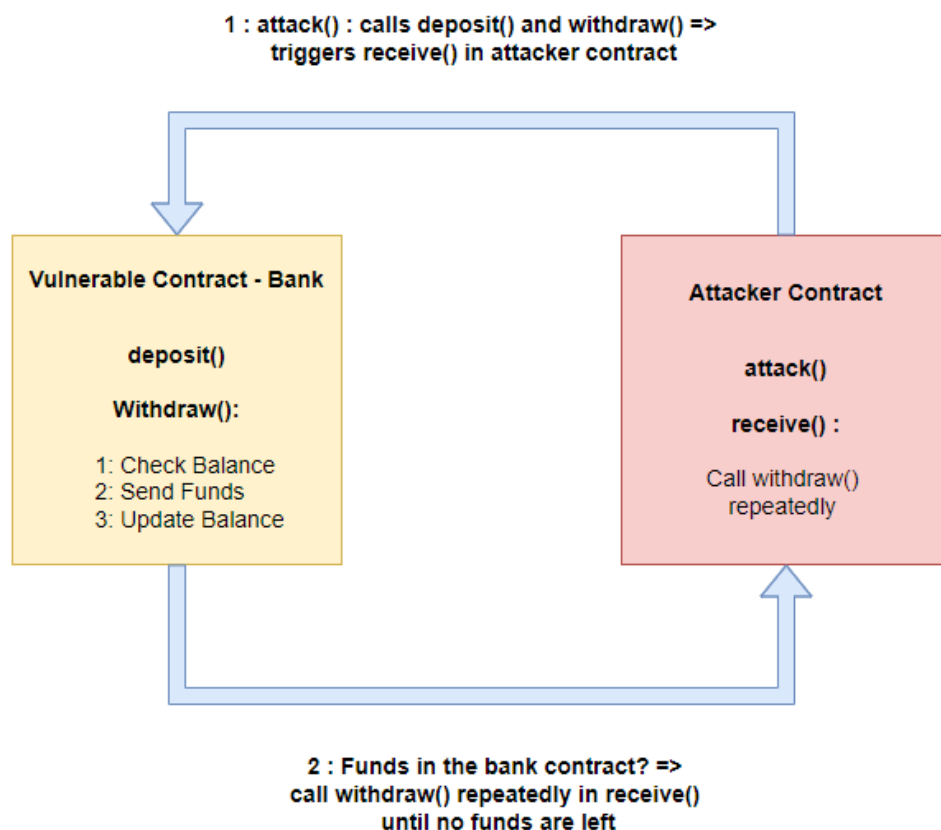
Protecting your code against a reentrancy attack

A reentrancy attack occurs between two smart contracts, where an attacking smart contract exploits the code in a vulnerable contract to drain it of its funds. The exploit works by having the attacking smart contract repeatedly call the withdraw function before the vulnerable smart contract has had time to update the balance.

This is only possible because of the order in which the smart contract is set up to handle transactions, with the vulnerable smart contract first checking the balance, then sending the funds, and then finally updating its balance. The time between sending the funds and updating the balance creates a window in which the attacking smart contract can make another call to withdraw its funds, and so the cycle continues until all the funds are drained.

An attacker makes a call on contract BANK (that has a reentrancy vulnerability) to transfer funds to the ATTACKER contract. Receiving the call, the BANK contract first checks to see that the attacker has the funds, then it transfers the funds to ATTACKER contract. Upon receiving the funds, the ATTACKER contract executes a fallback function which calls back into the BANK contract before it is able to update its balance, thus restarting the process.

The most infamous case of a reentrancy attack occurred in 2016 when Ethereum's DAO (decentralized autonomous organization) was hacked for an equivalent of 3.6 M ETH (\$60 million dollars). Ethereum's DAO was a project designed to act as a venture capital firm where members in the network could vote on initiatives to invest in (crowdfunding). In order to regain those funds, a hard fork was initiated. The original blockchain is now Ethereum Classic.



Reentrancy Attack Project

Requirements:

Bank Contract:

- Manage balance of users
- Deposit funds
- Withdraw funds
 - Deposited amount must be > 0
 - Transfer funds to user
 - Update user balance in contract

Attacker Contract:

- Create Bank contract in constructor
- Attack Bank contract: call deposit() and withdraw()
- receive(): check if funds left in Bank contract \Rightarrow call withdraw

Exercise:

Protecting the Bank contract:

- Inherit Bank contract from ReentrancyGuard (OpenZeppelin)
- Add nonReentrant modifier to withdraw function
- Apply the Check-Effects-Interaction pattern
- By using transfer instead of call this attack would not have been possible. However, the transfer method limits gas consumption in the fallback function to 2300 units of gas \Rightarrow this could break some existing contracts, because the gas cost may change for various opcodes, therefore it is no longer recommended to use the transfer method.

Checks-Effects-Interaction Pattern

This is the general guideline when coding any smart contract function. You start with validating input data. Next, you can make changes to the smart contract's state and finally interact with other smart contracts (for example, transferring funds). Interacting with other contracts should always be the last step in your function as this includes handing over the control to another contract.

DAO - Decentralized Autonomous Organization

What is a DAO

A DAO, or “Decentralized Autonomous Organization,” is a community-led entity with no central authority. It is fully autonomous and transparent: smart contracts execute the agreed upon decisions, and at any point, proposals, voting, and even the very code itself can be publicly audited. A DAO is governed entirely by its individual members who collectively make critical decisions.

The rules of the DAO are established by a core team of community members and they are enforced by smart contracts.

Community members create proposals about various operations and then they vote on each proposal. Proposals that achieve some predefined level of consensus are then accepted and enforced by smart contracts.

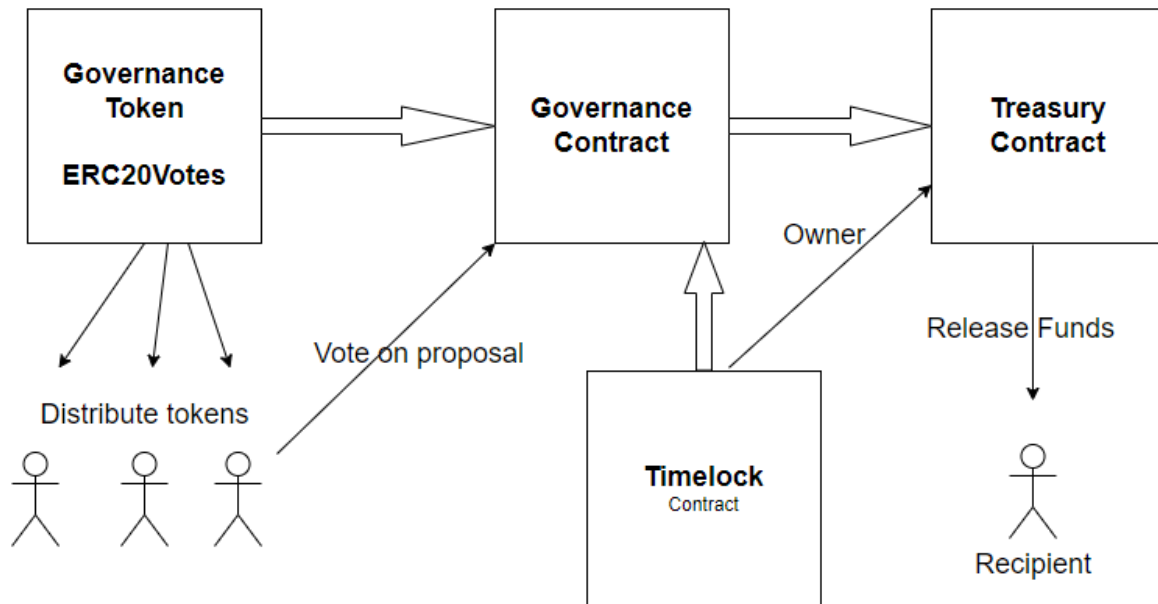
How does a DAO work?

Once these rules are formally written onto the blockchain (via smart contracts), the DAO needs to figure out how to receive funding. This is typically achieved through token issuance, by which the protocol sells tokens to raise funds and fill the DAO treasury.

In return for their fiat, token holders are given certain voting rights, usually proportional to their holdings. Once funding is completed, the DAO is ready for deployment. Proposals can be created, members can vote and successful proposals can be executed.

In certain protocols such as Compound, token holders can vote to distribute treasury funds towards bug fixes and system upgrades. This approach also allows developers to join ad hoc and receive compensation for their work.

Project Contracts:



- Distribute governance tokens - via crowd sale or other means
- Create a proposal on the governance contract - for example: a developer will receive a certain amount of tokens from the treasury upon completion of a specific work task
- Governance token holders can vote on that proposal
- If the vote is successful (conditions are encoded in the governance contract), the proposal is executed and the funds are sent to the recipient
- The timelock (owner of treasury contract) enforces a delay after which the funds are released after a successful vote

Governance token - ERC20Votes:

Extension of ERC20 to support Compound-like voting and delegation. This extension keeps a history (checkpoints) of each account's vote power.

Interface:

checkpoints(address account, uint32 pos) → struct ERC20Votes.Checkpoint
Get the pos-th checkpoint for account { uint32 fromBlock; uint224 votes; }

delegates(address account) → address
Get the address account is currently delegating to

delegate(address delegatee)

Delegate votes from the sender to delegatee

getVotes(address account) → uint256

Gets the current votes balance for account

ERC20Permit (used by ERC20Votes)

Allows approvals to be made via signatures. Adds the permit method, which can be used to change an account's ERC20 allowance by presenting a message signed by the account. The token holder account doesn't need to send a transaction, and thus is not required to hold Ether at all.

Treasury Contract:

The treasury contract holds the funds for the proposal. When the vote passes, the Timelock contract calls the "releaseFunds" function to transfer those funds to the receiver

MAYBE: Make this more generic, so we have only one Treasury for all proposals => don't specify funds and payee in constructor

Timelock Contract - TimelockController:

<https://docs.openzeppelin.com/contracts/4.x/api/governance#TimelockController>
<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/governance/TimelockController.sol>

It is good practice to add a timelock to governance decisions. This allows users to exit the system if they disagree with a decision before it is executed. When using a timelock, it is the timelock that will execute proposals. When set as the owner of an Ownable smart contract, it enforces a timelock on all onlyOwner operations.

The TimelockController uses an AccessControl setup:

- The Proposer role is in charge of queueing operations: this is the role the Governor instance should be granted, and it should likely be the only proposer in the system.
- The Executor role is in charge of executing already available operations.
- The Admin role can grant and revoke the two previous roles. This role will be granted automatically to both deployer and timelock, but should be renounced by the deployer after setup.

An operation is a transaction (or a set of transactions) that is managed by the timelock. It has to be scheduled (queued) by a proposer and executed by an executor. The timelock enforces a minimum delay between the proposition (queuing) and the execution

Operation status:

- Unset: An operation that is not part of the timelock mechanism.
- Pending: An operation that has been scheduled, before the timer expires.
- Ready: An operation that has been scheduled, after the timer expires.
- Done: An operation that has been executed.

Timelocked operations are identified by a unique id and follow a specific lifecycle: Unset → Pending → Ready → Done

By calling `schedule`, a proposer moves the operation from the Unset to the Pending state. This starts a timer. Once the timer expires, the operation automatically gets the Ready state. At this point, it can be executed. By calling `execute`, an executor triggers the operation's underlying transactions and moves it to the Done state.

Interface:

constructor(uint256 minDelay, address[] proposers, address[] executors)

Initializes the contract with a given minDelay, and a list of initial proposers and executors.

isOperationPending(bytes32 id) → bool pending

Returns whether an operation is pending or not.

isOperationDone(bytes32 id) → bool done

Returns whether an operation is done or not.

schedule(address target, uint256 value, bytes data...)

Schedule an operation containing a single transaction.

cancel(bytes32 id)

Cancel an operation.

execute(address target, uint256 value, bytes payload...)

Execute an (ready) operation containing a single transaction.

Governor Contract:

The core logic is given by the Governor contract, but we still need to choose: 1) how voting power is determined, 2) how many votes are needed for quorum, 3) what options people have when casting a vote and how those votes are counted, and 4) what type of token should be used to vote.

For 1) we will use the **GovernorVotes** module, which determines the voting power of an account based on the token balance they hold. This module requires as a constructor parameter the address of the token.

For 2) we will use **GovernorVotesQuorumFraction** which defines quorum as a percentage of the total supply at the block a proposal's voting power is retrieved. This requires a constructor parameter to set the percentage. Most Governors nowadays use 4%, so we will initialize the module with parameter 4.

For 3) we will use **GovernorCountingSimple**, a module that offers 3 options to voters: For, Against, and Abstain, and where only For and Abstain votes are counted towards quorum.

We also use **GovernorTimelockControl**: Connects with an instance of a TimelockController. A Timelock adds a delay for the execution of governance decisions. The workflow is extended to require a queue step before execution. Proposals are executed by the external timelock contract.

Besides these modules, Governor itself has some parameters we must set:

votingDelay: Delay (in number of blocks) from the creation of a proposal until voting starts.

votingPeriod: Delay (in number of blocks) from the creation of a proposal until voting ends.

quorum: Quorum required for a proposal to be successful.

Interface:

state(uint256 proposalId) → enum IGovernor.ProposalState

Current state of a proposal

getVotes(address account, uint256 blockNumber) → uint256

Voting power of an account at a specific blockNumber.

propose(address[] targets, uint256[] values, bytes[] calldatas, string description) → uint256 proposalId

Create a new proposal.

execute(address[] targets, uint256[] values, bytes[] calldatas, bytes32 descriptionHash) → uint256 proposalId

Execute a successful proposal.

castVote(uint256 proposalId, uint8 support) → uint256 balance

Cast a vote

GovernorCountingSimple

proposalVotes(uint256 proposalId) → uint256 againstVotes, uint256 forVotes, uint256 abstainVotes

Accessor to the internal vote counts.

Creating a proposal:

A proposal is a sequence of actions that the Governor contract will perform if it passes. Each action consists of a target address (the address of the smart contract that the timelock should operate on), calldata encoding a function call, and an amount of ETH to include. Additionally, a proposal includes a human-readable description.

Let's say we want to create a proposal to give a team a grant, in the form of ERC20 tokens from the governance treasury. This proposal will consist of a single action where the target is the ERC20 token, calldata is the encoded function call *transfer(<team wallet>, <grant amount>)*, and with 0 ETH attached.

After setting up the parameters of the proposal, we call the propose function of the governor. We pass in three arrays corresponding to the list of targets, the list of values, and the list of calldatas. In our case we have only one action:

```
await governor.propose([tokenAddress], [0], [transferCalldata], "Proposal #1: Give grant to team");
```

Once a proposal is active, delegates can cast their vote. If a token holder wants to participate, they can become a delegate themselves by self-delegating their voting power.

Once the voting period is over, if quorum was reached (enough voting power participated) and the majority voted in favor, the proposal is considered successful and can proceed to be executed.

If a timelock was set up, the first step to execution is queueing: *await governor.queue(...*

This will cause the governor to interact with the timelock contract and queue the actions for execution after the required delay. After enough time has passed (according to the timelock parameters), the proposal can be executed. If there was no timelock to begin with, this step can be ran immediately after the proposal succeeds. Executing the proposal will transfer the ERC20 tokens to the chosen recipient: *await governor.execute(...*

The DAO Project

Requirements:

Create the required contracts:

- GovToken: ERC20Votes - provide name, symbol and initial supply => mint initial supply to token owner address
- Treasury: Ownable contract, holds the funds for a specific proposal, allow transfer of funds to payee
- Timelock: TimelockController - provide minDelay, proposer(s) and executor(s)
- Governance: Governor, GovernorCountingSimple, GovernorVotes, GovernorVotesQuorumFraction, GovernorTimelockControl - provide voting delay and voting period

Create the deployment script (already done):

- Signers for executor, proposer, payee and 5 voters
- Deploy governance token: initial supply: 1000
- Transfer 50 governance tokens to each voter
- Deploy timelock contract: minDelay = 0
- Deploy governance contract: quorum = 5, votingDelay = 0, votingPeriod = 55
- Timelock grants proposer and executor roles to the governance contract
- Deploy treasury contract: specify payee address and transfer proposal funds (5 ETH), transfer ownership to timelock

Create the proposal script (already done):

- Self-delegate voting rights to 5 voters
- Display initial treasury and payee balance
- Encode the releaseFunds function from the treasury contract => required for the proposal
- Create a proposal: There is one action => target: address of treasury, value: 0, calldata: encoded "releaseFunds" function
- Display the proposal Id from the ProposalCreated event
- Display the state of the proposal
- Display the blocks for the proposal creation and the proposal deadline
- Cast the votes
- Display the number of votes (for, against and abstain)
- Queue the proposal
- Execute the proposal
- Display the state of the proposal
- Display final treasury and payee balance