ETHEREUM 2.0 MASTERY PROGRAM

Instructor: Raja Rizwan Saleem

MODULE-2

BLOCKCHAIN AND SMART CONTRACT BASICS

Raja Rizwan Saleem
Lead Blockchain Trainer

Class-14

Solidity Imports

Imports in Solidity

Single File Import:

- Import the entire file for access to all its contracts, libraries, and interfaces.
- Syntax: `import "./FileName.sol";`

Importing Specific Symbols:

- Import only specific contracts, libraries, or interfaces from a file.
- Syntax: `import { ContractName } from "./FileName.sol";`

Importing Multiple Symbols:

- Import multiple specific symbols from a single file.
- Syntax:
```
import { Contract1, Contract2 } from
"./FileName.sol";
```

Imports in Solidity

Global Import (Wildcard Import):

- Import everything from a file using the * wildcard.
- Syntax: `import * as AliasName from "./FileName.sol";`

Aliasing Imports:

- Use an alias for imported contracts, libraries, or interfaces to avoid naming conflicts or for convenience.
- Syntax: `import { ContractName as AliasName } from "./FileName.sol";`

Importing from Node Modules:

- Import contracts or libraries from external dependencies installed via `npm` or `yarn`.
- Syntax: `import "package-name/FileName.sol";`

Imports in Solidity

Relative Path Import:

- Import files using relative paths, which are paths relative to the current file's location.
- Syntax: `import "../directory/FileName.sol";`

Importing from GitHub:

- Directly import contracts or libraries from a GitHub repository using a raw URL.
- Syntax: `import "https://github.com/UserName/RepoName/blob/branch/contracts/FileName.sol";`

Importing Interfaces:

- Import interfaces from another file to implement them in your contract.
- Syntax: `import "./InterfaceName.sol";`

Import a Single File

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

// Importing a single file
import "./Parent.sol";

contract Child is Parent {
    function getValue() public view returns (uint) {
        return value;
    }
}
```

Import a Specific symbols

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

// Importing specific symbols (e.g., specific contract)
import { Parent } from "./Parent.sol";

contract Child is Parent {
    function getValue() public view returns (uint) {
        return value;
    }
}
```

Import a Multiple symbols

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

// Importing multiple symbols
import { Parent, Helper } from "./ParentAndHelper.sol";

contract Child is Parent {
    function useHelper(uint _value) public {
        Helper.setHelperValue(_value);
    }

    function getHelperValue() public view returns (uint) {
        return Helper.helperValue();
    }
}
```

Importing from Github

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

// Importing from GitHub
import
"https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts
/token/ERC20/ERC20.sol" ;

contract MyToken is ERC20 {
        constructor() ERC20("MyToken", "MTK") {
                _mint(msg.sender, 1000 * 10 ** 18);
    } }
```

OPENZEPPELIN + REMIX + BNB SMART CHAIN

ERC-20, ERC-721 TOKENS

Solidity Smart Contract refresher
Tokens and token standards: ERC-20 and ERC-721
OpenZeppelin

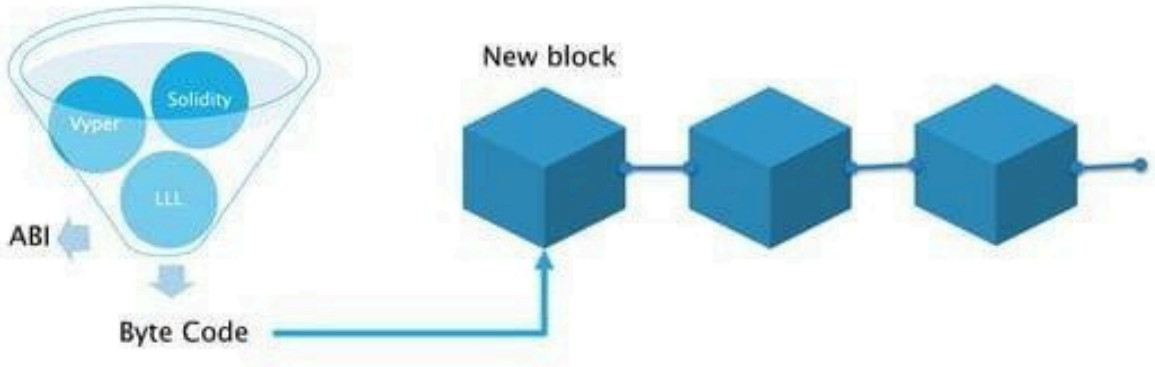
5/6/2021

SMART CONTRACTS

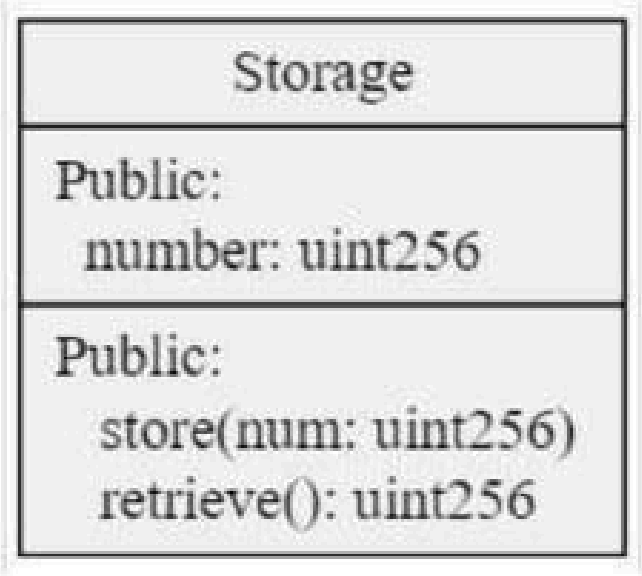
Smart contracts in 5 minutes or less

SMART CONTRACTS

SMART CONTRCTS ON BLOCKCHAIN



SMART CONTRACT EXAMPLE



SMART CONTRACT CODE EXAMPLE

```
pragma solidity >=0.7.0 <0.9.0;

/**
 * @title Storage
 * @dev Store & retrieve value in a variable
 */
contract Storage {

    uint256 number;

    /**
     * @dev Store value in variable
     * @param num value to store
     */
    function store(uint256 num) public {
        number = num;
    }

    /**
     * @dev Return value
     * @return value of 'number'
     */
    function retrieve() public view returns (uint256){
        return number;
    }
}
```

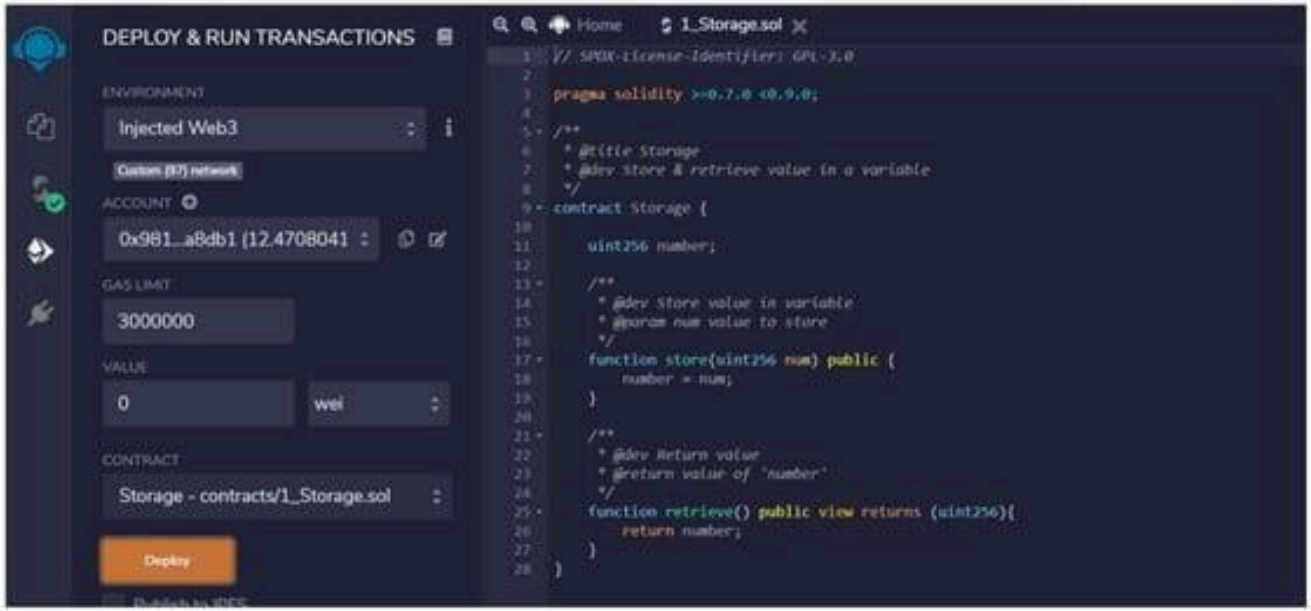
BYTECODE/OPCODE

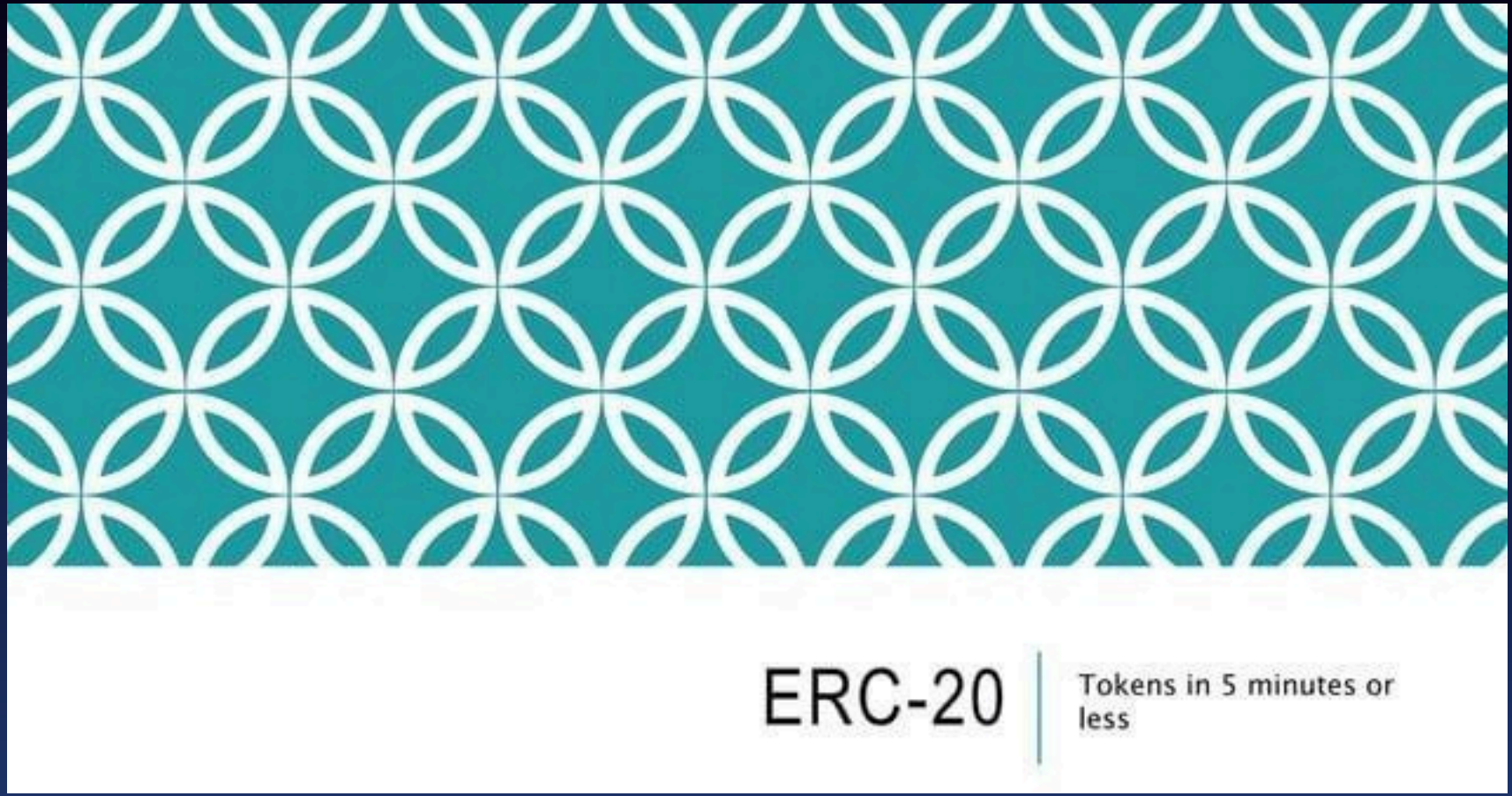
PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH1 0xF JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP PUSH1 0xAB DUP1 PUSH2 0x1E PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN INVALID PUSH1 0x80 PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH1 0xF JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP PUSH1 0x4 CALLDATASIZE LT PUSH1 0x32 JUMPI PUSH1 0x0 CALLDATALOAD PUSH1 0xE0 SHR DUP1 PUSH4 0x2E64CEC1 EQ PUSH1 0x37 JUMPI DUP1 PUSH4 0x6057361D EQ PUSH1 0x4C JUMPI JUMPDEST PUSH1 0x0 DUP1 REVERT JUMPDEST PUSH1 0x0 SLOAD PUSH1 0x40 MLOAD SWAP1 DUP2 MSTORE PUSH1 0x20 ADD PUSH1 0x40 MLOAD DUP1 SWAP2 SUB SWAP1 RETURN JUMPDEST PUSH1 0x5C PUSH1 0x57 CALLDATASIZE PUSH1 0x4 PUSH1 0x5E JUMP JUMPDEST PUSH1 0x0 SSTORE JUMP JUMPDEST STOP JUMPDEST PUSH1 0x0 PUSH1 0x20 DUP3 DUP5 SUB SLT ISZERO PUSH1 0x6E JUMPI DUP1 DUP2 REVERT JUMPDEST POP CALLDATALOAD SWAP2 SWAP1 POP JUMP INVALID LOG2 PUSH5 0x6970667358 0x22 SLT KECCAK256 0xE1 0xB8 0xD0 DUP8 0xE8 DUP4 0x22 PUSH28 0xC716E567D1D88DE9038140C655CE12488AFFAA4436CAA74264736F6C PUSH4 0x43000803 STOP CALLER

ABI

```
[...
    {
        "inputs": [
            {
                "internalType": "uint256",
                "name": "num",
                "type": "uint256"
            }
        ],
        "name": "store",
        "outputs": [],
        "stateMutability": "nonpayable",
        "type": "function"
    }
]
```

DEMO


DEPLOY & RUN TRANSACTIONS
ENVIRONMENT
Injected Web3
Custom (97) network
ACCOUNT
0x981...a8db1 (12.4708041
GAS LIMIT
3000000
VALUE
0
wei
CONTRACT
Storage - contracts/1_Storage.sol
Deploy
Home
1_Storage.sol
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
/**
* @title Storage
* @dev Store & retrieve value in a variable
*/
contract Storage {
uint256 number;
/**
* @dev Store value in variable
* @param num value to store
*/
function store(uint256 num) public {
number = num;
}
/**
* @dev Return value
* @return value of 'number'
*/
function retrieve() public view returns (uint256){
return number;
}
}


ERC-20
Tokens in 5 minutes or less

ERC-20

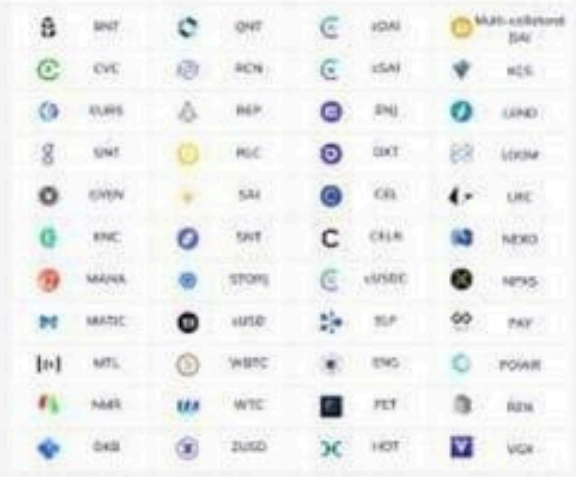
ERC-20 has emerged as the technical standard; it is used for all smart contracts on the Ethereum blockchain for token implementation and provides a list of rules that all Ethereum-based tokens must follow.

FUNCTIONS

- totalSupply()
- balanceOf(account)
- transfer(recipient, amount)
- allowance(owner, spender)
- approve(spender, amount)
- transferFrom(sender, recipient, amount)

EVENTS

- Transfer(from, to, value)
- Approval(owner, spender, value)



ERC-20 IS THE MOST POPULAR TOKEN STANDARD

A total of 353,518 Token Contracts found

First < Page 1 of 10

#	Token	Price	Change (%)	Volume (24H)	Market Cap
1	BNB (BNB) Binance aims to build a world-class crypto exchange, powering the future of crypto finance.	$637.84 0.010954 Btc 0.224246 Eth	4.64%	$6,924,042,398.00	$98,561,716,341.00
2	Tether USD (USDT) Tether gives you the joint benefits of open blockchain technology and traditional currency by converting your cash into a stable digital currency equivalent.	$0.9998 0.000017 Btc 0.000351 Eth	-0.26%	$114,438,747,662.00	$51,670,960,958.00
3	Uniswap (UNI) UNI token served as governance token for Uniswap protocol with 1 billion UNI have been minted at genesis. 60% of the UNI genesis supply is allocated to Uniswap community members and remaining for team, investors and advisors.	$41.01 0.000704 Btc 0.014416 Eth	1.54%	$1,216,026,354.00	$21,284,378,777.00

COMMON ERC-20 EXTENSIONS – NAME,SYSMBOL, DECIMALS

- constructor(name, symbol, decimals)
- name()
- symbol()
- decimals()


OPENZEPPELIN
4.0

OPENZEPPELIN CONTRACTS

OpenZeppelin

OpenZeppelin Contracts is a library for secure smart contract development. It provides implementations of standards like ERC20 and ERC721 which you can deploy as-is or extend to suit your needs, as well as Solidity components to build custom contracts and more complex decentralized systems.

OPENZEPPELIN BENEFITS

Open-source (MIT License)

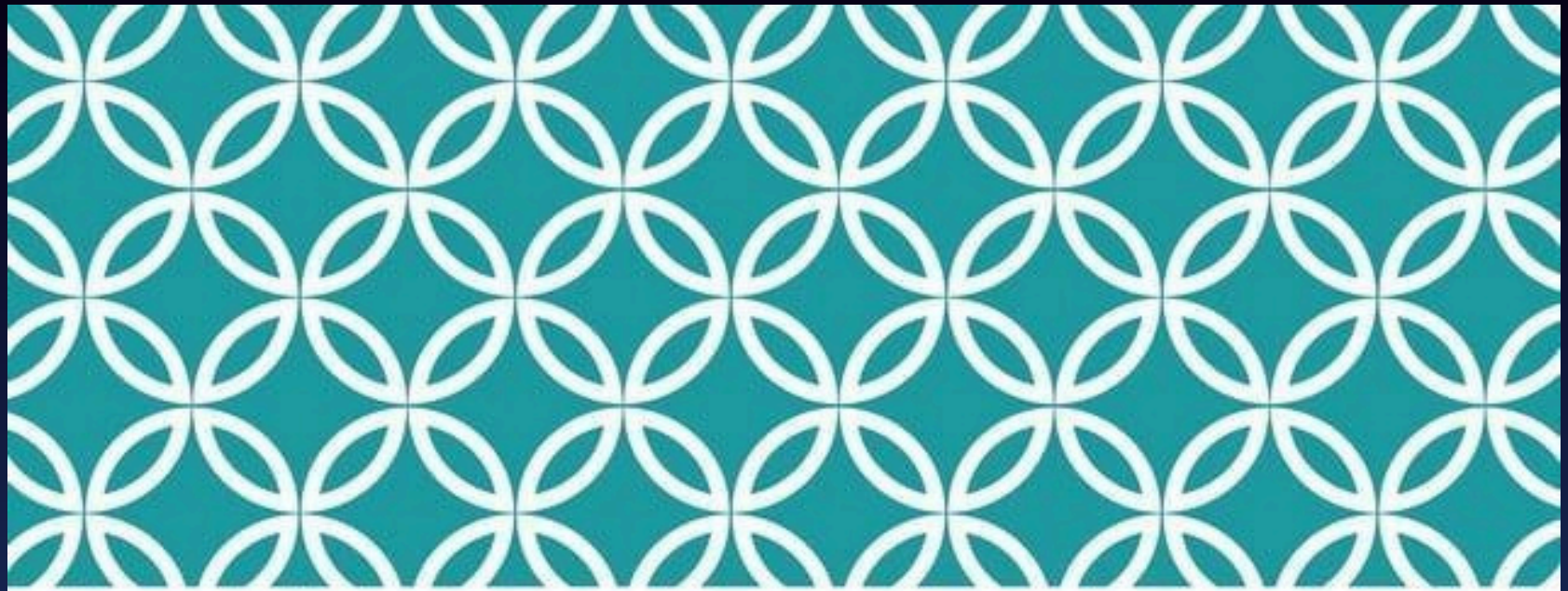
Implementations of standards like ERC20 and ERC721.

Flexible role-based permissioning scheme.

Reusable Solidity components to build custom contracts and complex decentralized systems.

First-class integration with the Gas Station Network for systems with no gas fees!

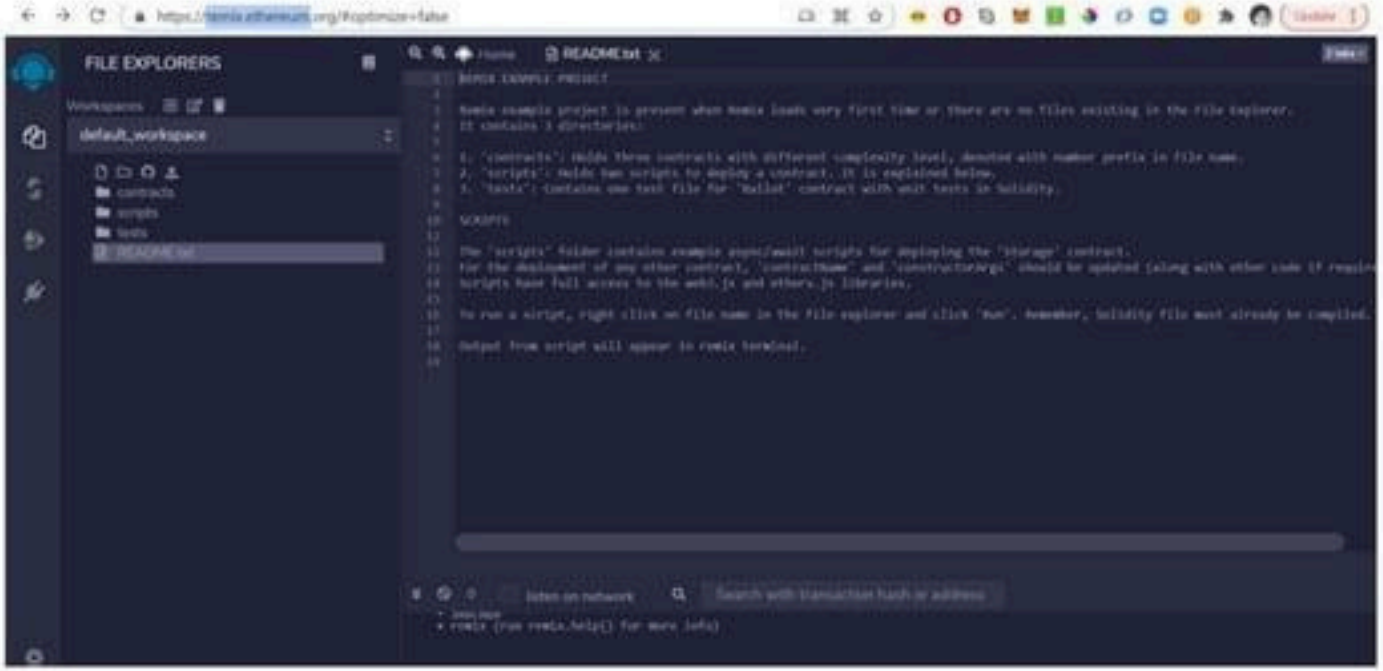
Audited by leading security firms



REMIX

IDE by Ethereum.org

REMIX IDE



WHY REMIX?

- Ethereum integration
- Blockchain emulator
- Remix can support both Solidity and Viper
- Many plugin related to smart contract development, testing, and documentation
- Visual debugger
- Selection of compiler version


BINANCE SMART CHAIN
IDE by Ethereum.org

BINANCE CHAINS

BINANCE

Binance Chain

Lunched April 2019

Enables users to send and receive BNB

Uses Tendermint BFT consensus

Users can use DEX

Binance Smart Chain

Lunched September 2020

Enables decentralized applications

EVM Compatible

Proof of Stake Authority (PoSA)

On-Chain Governance

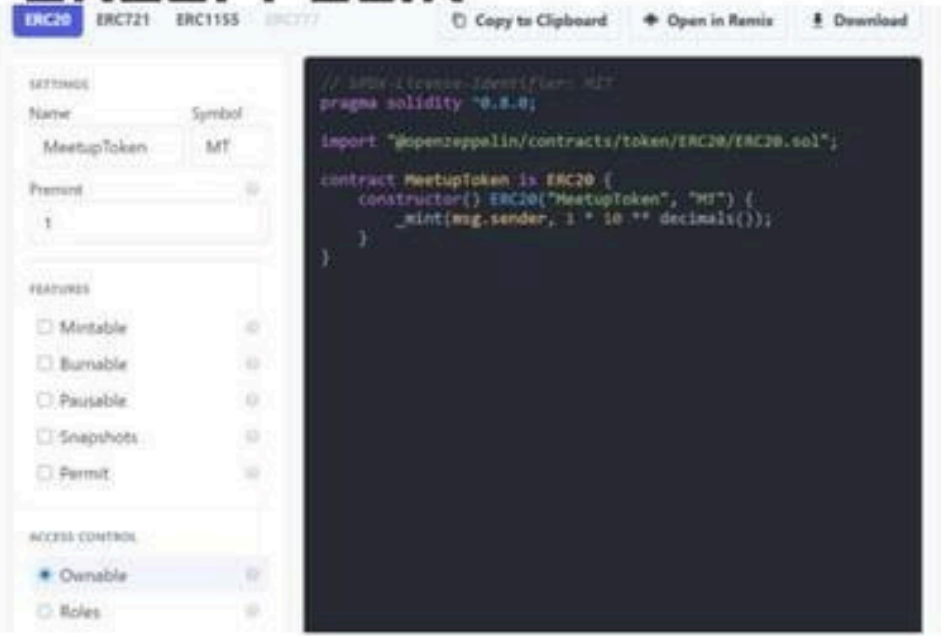
Interoperability and Cross-Chain Transfer (using BNB)


BINANCE
SMART CHAIN
DeFi
Landscape
—
Nov. 19 2020
WALLETS
Binance Chain Wallet
BitKeep
MathWallet
SafePal
Trust Wallet
TokenPocket
INFRASTRUCTURE
ANKR
Band Protocol
BSCscan
ChainLink
Gas Station Network
Snapshot
BTC
anyBTC
Binance BTC
renBTC
STABLECOINS
BUSD
PAX
QIAN
VAI (Venus Stablecoin)
CROSS CHAIN
AnySwap
Binance Bridge
renVM
EXCHANGE & LIQUIDITY
BakerySwap
Bounce.Finance
bStable.finance
BurgerSwap
DODO
Equator.Finance
PancakeSwap
Spartan Protocol
StableXSwap
UniFi
CREDIT & LENDING
7up.finance
Cream.finance
ForTube
Venus
CokeFinance
YIELD FARMING & AGGREGATORS
BeefyFinance
BFIs.finance
Defi.money (YFII)
Dego
fry.World
ANALYTICS
BitQuery
CoinMarketCap
DefiStation
DappRadar
PARSIQ
KYC & IDENTITY
Ontology
INSURANCE
Certik
PAYMENT
SWFT

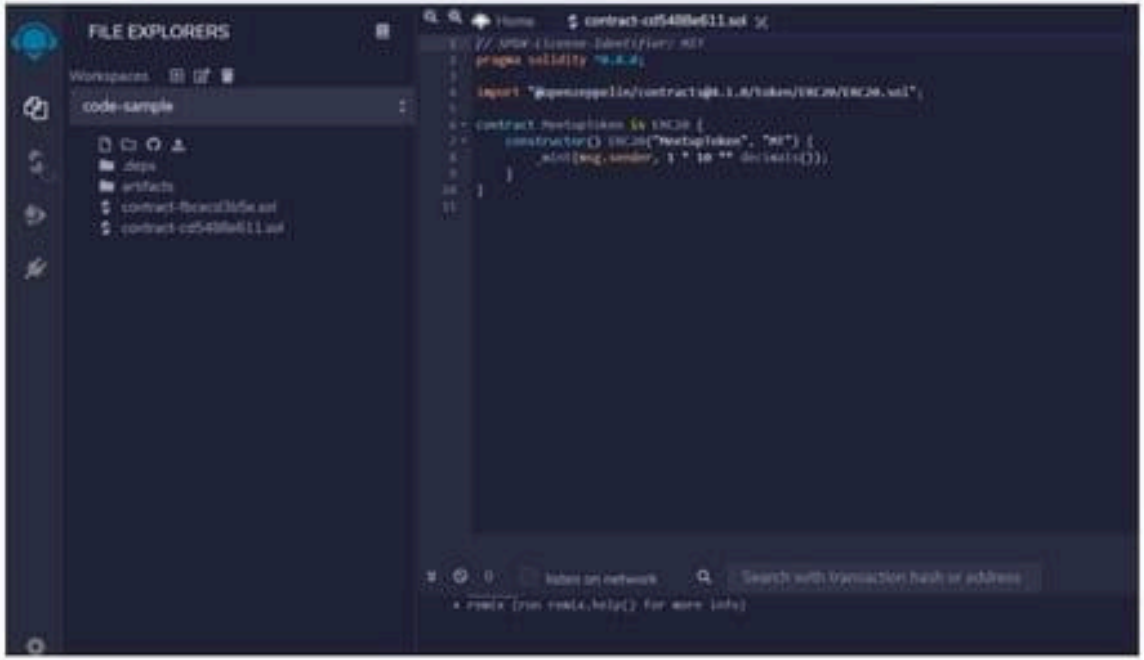
ALL TOGETHER NOW

OpenZeppelin + Remix +
BNB Smart Chain

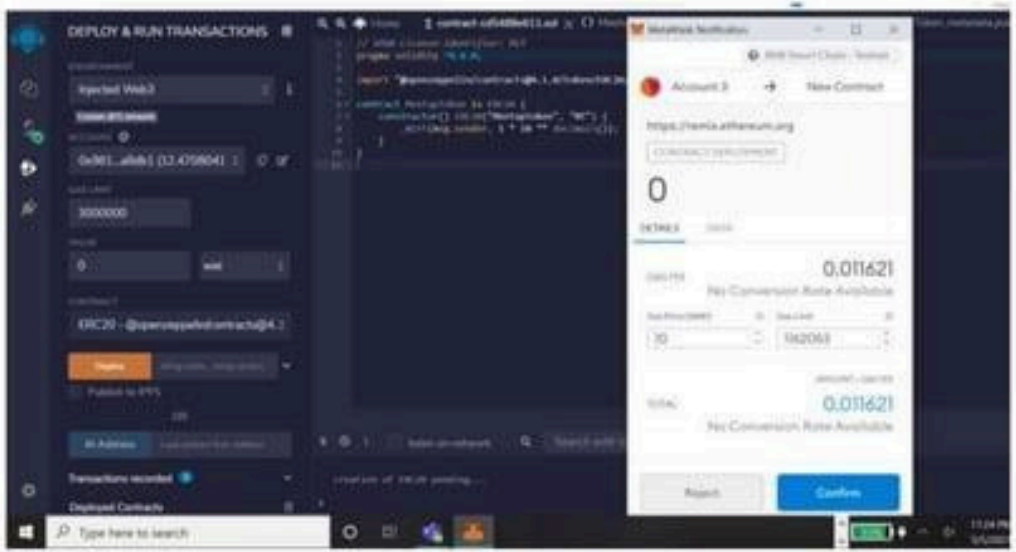
1. CREATE ERC-20 TOKEN WITH **OPENZEPPELIN**



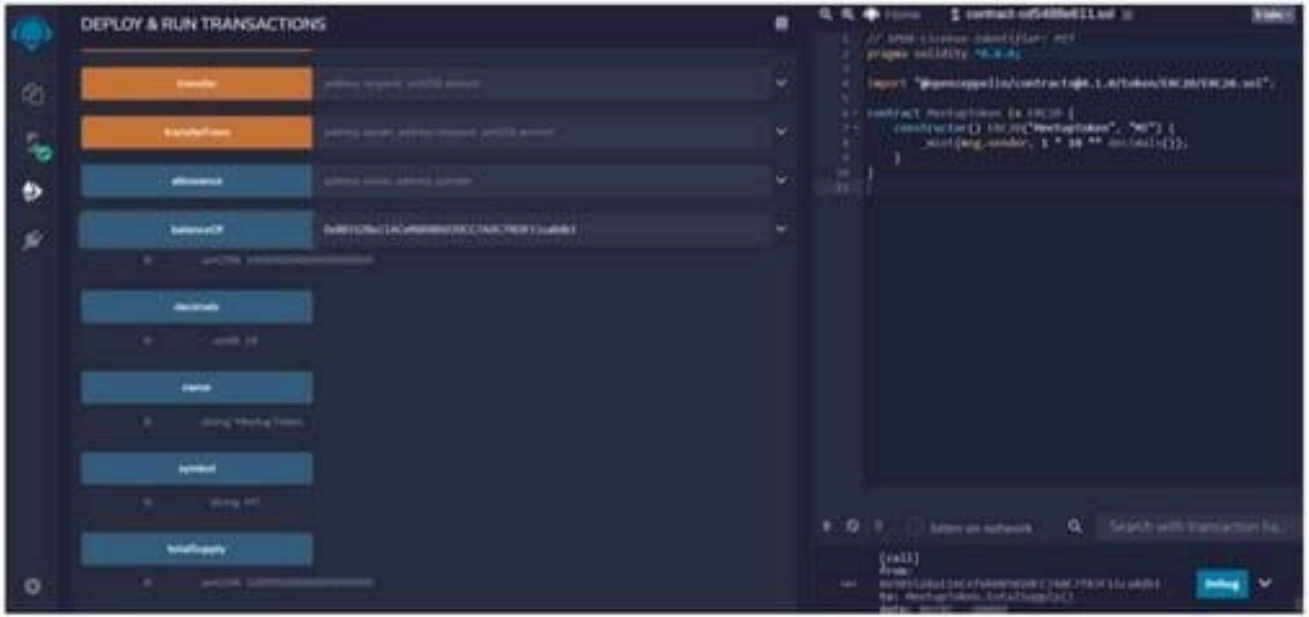
2. OPEN CODE IN **REMIX IDE**



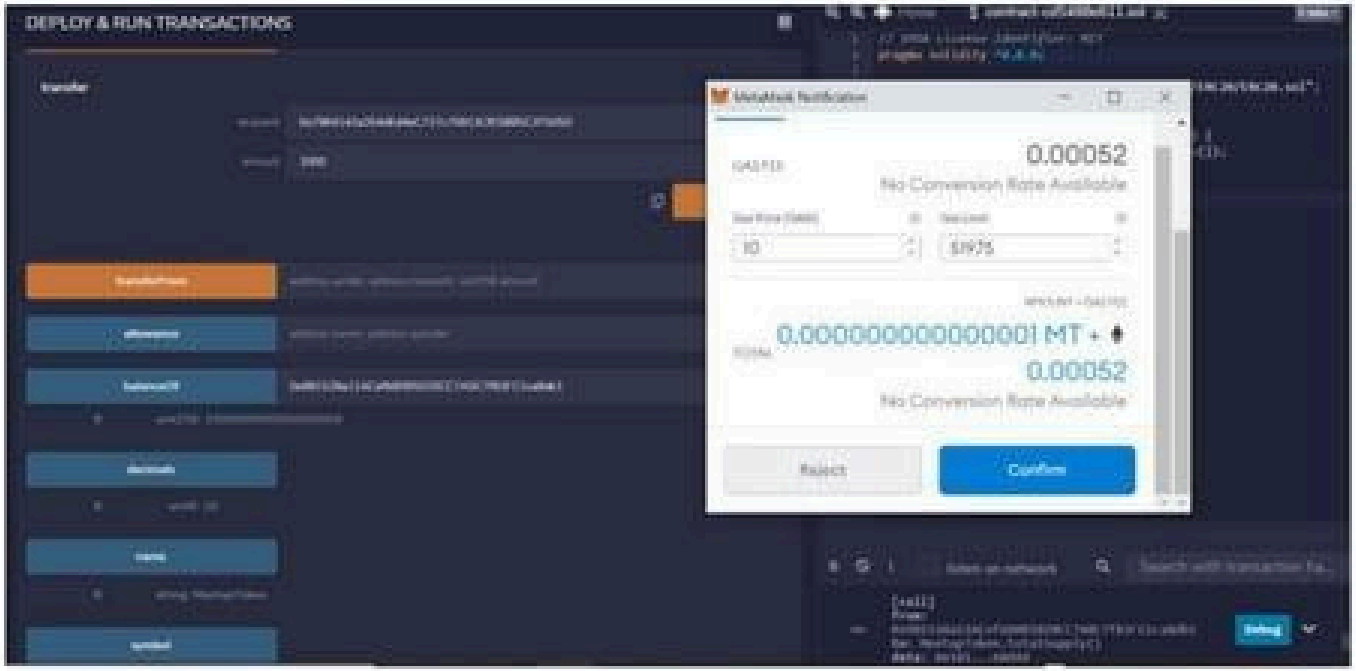
4. DEPLOY


DEPLOY & RUN TRANSACTIONS
Injected Web3
0
0.011621
0.011621
Confirm
Type here to search

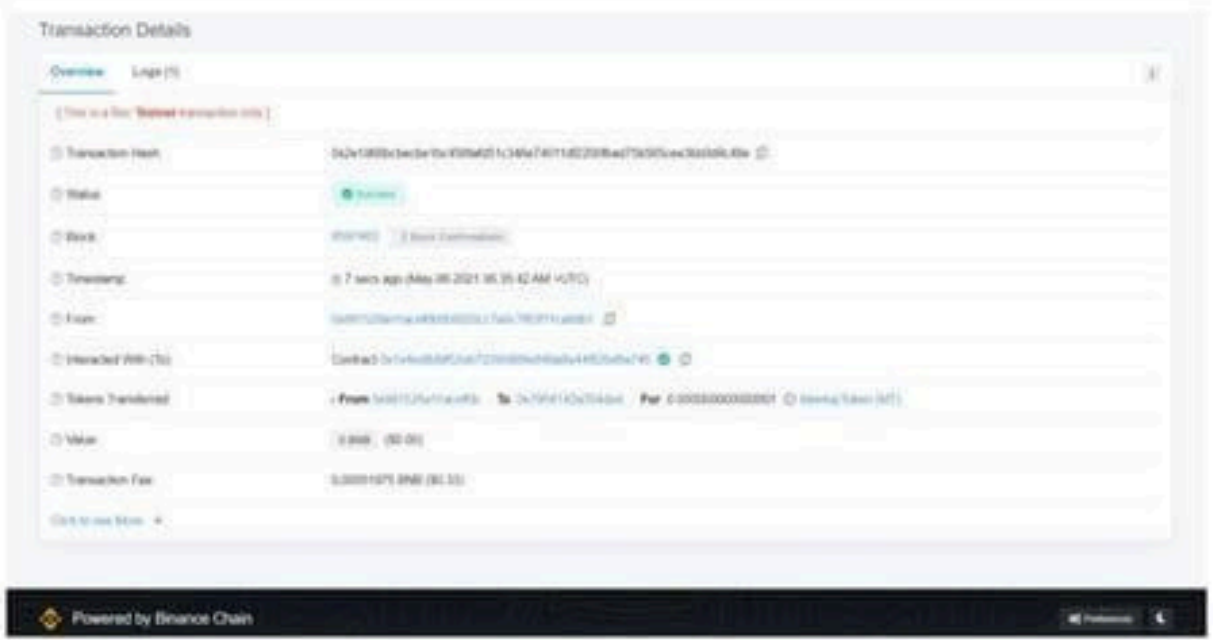
5. CHECK



6. SEND TOKEN TO ANOTHER ACCOUNT



7. CHECK ON BSCSCAN



Smart Contract Use Cases

Smart Contract Use Cases

- From simple transactions to complex endeavors, these self-executing contracts remove the middlemen and create independence.
- No matter the industry or scenario, the middlemen always:
- Want a cut. With an automated smart contract, you do not need to trust or pay middlemen because they are not needed. This streamlines the process and can make smart contracts cost-effective.
- The entire system is essentially trustless. You do not need to trust any other parties, such as brokers or lawyers, to enforce or carry out the transaction.
- This means smart contracts are fast and disruption-free.
- Blockchain technology powering smart contracts creates immutable data that nobody can change.
- Encrypted data adds a layer of security to the transactions.

Smart Contract Use Cases

- Various industries are beginning to recognize the versatility of smart contract technology.
- When it comes to real-world examples, particularly with property ownership and financial services, the sky's the limit.
- Smart contracts run on a public ledger, making transactions visible to everyone on the network, ensuring transparency.
- Transactions on a distributed ledger cannot be changed or deleted.
- Smart contracts can handle:
  - Simple transactions.
  - Detailed transactions involving multiple parties.
- Blockchain technology allows the use of coding languages like Solidity to craft transactions on platforms such as:
  - Ethereum Virtual Machine (EVM).
  - Hedera.
  - Other blockchain platforms.
- Once created, a smart contract can be reused and connected to other transactions.

Smart Contract Use Cases

Smart Contract Use Cases

Smart Contract Use Cases

Smart Contract Use Cases

Smart Contract Use Cases

Smart Contract Use Cases

A voting system implemented through a smart contract is a transparent and tamper-proof method for conducting polls, decision-making processes, or elections using blockchain technology. This approach guarantees the integrity of the voting process by eliminating the need for a central authority or intermediary, such as a government or trusted organization. In this explanation, we will explore how a voting system works through a smart contract, detailing its components and functionalities.

Smart Contract Use Cases

Smart Contract Creation:

- A smart contract is a self-executing program that runs on a blockchain (e.g., Ethereum). It contains the rules and logic for the voting process.
- An entity, like an organization or a group of individuals, creates the smart contract and deploys it on a blockchain.

Voter Registration:

- To participate in the vote, individuals must register their digital identities on the blockchain. This can be done by creating a digital wallet and verifying their identity, ensuring that only eligible voters can participate.

Voting Process:

- When the voting period begins, registered voters can interact with the smart contract through a user-friendly interface, such as a website or mobile app.
- Voters submit their choices by signing a transaction with their private keys, which are linked to their digital identities.

Vote Recording:

- The smart contract records each vote on the blockchain in an immutable and transparent manner.
- The blockchain's decentralized nature ensures that once a vote is recorded, it cannot be altered or deleted, providing a high level of security and transparency.

Smart Contract Use Cases

Counting and Verification:

- The smart contract automatically tallies the votes once the voting period ends, based on the predefined rules and criteria.
- Anyone can verify the vote count by inspecting the blockchain's public ledger, ensuring the accuracy of the results. Result Announcement:
- After the vote is counted, the smart contract can publish the results directly on the blockchain or through a user interface.
- Voters can verify that their votes were included in the final tally, adding transparency and trust to the process. Security and Tamper Resistance:
- Blockchain technology ensures that the voting process is highly secure and resistant to tampering. Once a vote is recorded on the blockchain, it becomes nearly impossible to alter or manipulate.

Accessibility and Convenience:

- Voters can participate in the voting process from anywhere with an internet connection, making it convenient and accessible for a broader audience.

Privacy and Anonymity:

- Depending on the design of the smart contract, voter identities can be kept private while still ensuring the integrity of the voting process.

Auditability:

- The entire voting process, including voter registration, vote casting, and result tabulation, can be audited by anyone with access to the blockchain's data, enhancing transparency and trust.

Smart Contract Use Cases -Voting Contract

A Voting Smart Contract enables a decentralized, transparent, and tamper-proof voting system on the blockchain. It ensures that only authorized users can vote and that votes are counted accurately.

Smart Contract Use Cases - Voting Contract

2. Key Functionalities

- **Candidate Management**
  - Add candidates
  - Retrieve candidate details
- **Voter Management**
  - Register voters
  - Ensure each voter can vote only once
- **Voting Process**
  - Cast vote
  - Prevent duplicate votes
- **Result Compilation**
  - Display vote count
  - Declare the winner

Smart Contract Use Cases - Voting Contract

3. Smart Contract Structure

A. State Variables

- `struct Candidate { uint id; string name; uint voteCount; }`
- `mapping(uint => Candidate) public candidates;`
- `uint public candidatesCount;`
- `mapping(address => bool) public hasVoted;`
- `address public electionAdmin;`

B. Constructor

- Set contract deployer as the `electionAdmin`
- Initialize candidates

Smart Contract Use Cases - Voting Contract

C. Functions

1. **addCandidate(string memory _name)**
   - Only `electionAdmin` can add candidates
   - Increment `candidatesCount`
2. **registerVoter(address _voter)**
   - Only `electionAdmin` can register voters
3. **vote(uint _candidateId)**
   - Ensure voter has not voted before
   - Increment candidate's `voteCount`
   - Mark voter as `hasVoted`
4. **getResults()**
   - Display all candidates and their votes
5. **declareWinner()**
   - Identify the candidate with the most votes

Smart Contract Use Cases - Voting Contract

4. Security Considerations

- Use `require()` to prevent unauthorized access
- Ensure valid candidate ID before voting
- Prevent double voting

---

5. Future Enhancements

- Add time-based voting restrictions
- Implement vote delegation
- Store voting results in an IPFS system for added transparency

Smart Contract Use Cases

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


contract Voting {
    address public owner;
    string public electionName;
    mapping(address => bool) public voters;
    mapping(string => uint256) public voteCounts;


    enum VoteStatus {
    NotVoted,
    Voted
    }

    struct Voter {
        VoteStatus status;
        string selectedCandidate;
```

Smart Contract Use Cases

```
struct Voter {
    VoteStatus status;
    string selectedCandidate;
}

mapping(address => Voter) public voterInfo;

event NewElection(string electionName);
event NewCandidate(string candidateName);
event VoteCast(address indexed voter, string candidate);
event ElectionResults(string candidate, uint256 voteCount);


modifier onlyOwner() {
    require(msg.sender == owner, "Only the owner can perform this action");
    _;
}
```

Smart Contract Use Cases

```
 constructor(string memory _electionName) {
     owner = msg.sender;
     electionName = _electionName;
     emit NewElection(_electionName);
}

function addVoter(address _voter) external onlyOwner {
     require(!voters[_voter], "Voter already added");
     voters[_voter] = true;
}

function addCandidate(string memory _candidateName) external onlyOwner {
require(voteCounts[_candidateName] == 0, "Candidate already added");
voteCounts[_candidateName] = 0;
emit NewCandidate(_candidateName);
}
```

Smart Contract Use Cases

```
    function vote(string memory _candidateName) external {
        require(voters[msg.sender], "You are not authorized to vote");
        require(voterInfo[msg.sender].status == VoteStatus.NotVoted, "You have already voted");

        voterInfo[msg.sender].status = VoteStatus.Voted;
        voterInfo[msg.sender].selectedCandidate = _candidateName;

        voteCounts[_candidateName]++;

        emit VoteCast(msg.sender, _candidateName);
    }

    function getVoterStatus(address _voter) external view returns (VoteStatus status, string memory
selectedCandidate) {
        status = voterInfo[_voter].status;
        selectedCandidate = voterInfo[_voter].selectedCandidate;
    }
```

Smart Contract Use Cases

```
    function getVoteCount(string memory _candidateName) external view returns (uint256) {
        return voteCounts[_candidateName];
    }

    function announceResults(string memory _candidateName) external onlyOwner {
        emit ElectionResults(_candidateName, voteCounts[_candidateName]);
    }
}
```

Smart Contract Use Cases

Contract Overview:

- The contract is named "Voting."
- It has an owner, who is the address that deployed the contract.
- The contract has a name for the election, specified during contract deployment. Data Structures:
- voters: A mapping of addresses to boolean values, indicating whether an address is authorized to vote. Only authorized voters can cast their votes.
- voteCounts: A mapping of candidate names (strings) to the number of votes they have received.
- voterInfo: A mapping of addresses to a Voter struct, which includes information about the voting status (whether they have voted or not) and the candidate they selected

Smart Contract Use Cases

Enum and Struct:

- VoteStatus: An enumeration that defines two possible states for a voter: NotVoted and Voted.
- Voter: A struct that holds information about a voter, including their voting status and the candidate they selected.

Events:

- NewElection: An event emitted when the contract is deployed to announce the name of the election.
- NewCandidate: An event emitted when a new candidate is added to the election.
- VoteCast: An event emitted when a voter casts their vote, indicating the voter's address and the candidate they voted for.
- ElectionResults: An event emitted to announce the results of the election for a specific candidate, including the candidate's name and the vote count they received.

Modifiers:

- onlyOwner: A modifier that restricts certain functions to be callable only by the owner of the contract. This ensures that only the owner can perform critical actions like adding voters and announcing results.

Smart Contract Use Cases

Constructor:

- The constructor is executed only once during contract deployment. It sets the owner and the name of the election and emits the NewElection event.

Functions:

- addVoter: Allows the owner to add authorized voters by specifying their Ethereum addresses.
- addCandidate: Allows the owner to add new candidates to the election by specifying their names.
- vote: Allows authorized voters to cast their votes for a specific candidate. It updates the voter's status and increments the vote count for the selected candidate.
- getVoterStatus: Retrieves the voting status and the selected candidate of a specific voter.
- getVoteCount: Retrieves the number of votes received by a specific candidate.
- announceResults: Allows the owner to announce the election results for a specific candidate by emitting the ElectionResults event.

https://techblog.geekyants.com/e-voting-via-blockchain-a-case-study

Smart Contract Use Cases

Smart Contract Use Cases

A clinical trials smart contract is a blockchain-based solution designed to streamline and optimize the management and execution of clinical trials in the pharmaceutical industry. By leveraging blockchain technology, it ensures transparency, security, and efficiency throughout the entire clinical trial process. In this discussion, we will explore the key components and detailed workings of clinical trials smart contracts.

Smart Contract Use Cases

Participant Enrollment and Informed Consent:

- The smart contract initiates the clinical trial by allowing potential participants to enroll securely. Participants provide their consent to join the trial, and this consent is recorded on the blockchain.
- Participant identity and personal information can be anonymized using cryptographic techniques to protect privacy while maintaining data integrity.

Trial Protocol and Criteria:

- The contract stores the trial protocol, including inclusion and exclusion criteria, treatment procedures, and trial duration.
- It ensures that participants meet the required criteria before they are enrolled in the trial, reducing the risk of bias.

Randomization and Allocation:

- Randomization algorithms within the smart contract allocate participants to different treatment groups or arms. This ensures unbiased assignment, enhancing the validity of the trial.
- The allocation results are recorded immutably on the blockchain, preventing any tampering with the randomization process.

Smart Contract Use Cases

Data Collection and Monitoring:

- Smart contracts facilitate the collection of clinical trial data in real-time from various sources, such as IoT devices, electronic health records, and wearable devices.
- Data is timestamped and securely stored on the blockchain, creating an auditable and tamper-resistant trail.
- Monitoring parameters can trigger alerts and notifications if predefined thresholds or adverse events occur during the trial.

Incentives and Compensation:

- Smart contracts can automate the distribution of incentives or compensation to participants, investigators, and other stakeholders based on predefined rules and milestones.
- This ensures that participants are fairly compensated and incentivized to complete the trial.

Smart Contract Use Cases

Data Privacy and Security:

- Blockchain's encryption and consensus mechanisms ensure that sensitive patient data remains private and secure while still being accessible to authorized parties.
- Participants have control over who can access their data through permissioned access. Transparency and Auditing:
- Every action and transaction related to the clinical trial is recorded on the blockchain, creating a transparent and auditable ledger.
- Auditors and regulatory authorities can easily verify the trial's integrity and compliance with regulations. Smart Contract Governance:
- Smart contracts can include governance features that allow trial administrators to make predefined changes, such as protocol amendments or participant additions/removals, while maintaining a record of these changes.

Smart Contract Use Cases

Interoperability:

- To enhance collaboration and data sharing, clinical trials smart contracts may be designed to interoperate with other healthcare systems and databases using standard protocols and APIs.

Regulatory Compliance:

- The smart contract can embed regulatory requirements and automatically enforce compliance throughout the trial, reducing the risk of regulatory issues and fines.

Trial Conclusion and Reporting:

- When the trial concludes, the smart contract can automatically generate and distribute reports summarizing the trial results, making the process more efficient and less error-prone.

Dispute Resolution:

- In cases of disputes or discrepancies, predefined dispute resolution mechanisms within the smart contract can facilitate fair and transparent resolution.

Smart Contract Use Cases - Clinical Trial

```
pragma solidity ^0.8.0;

contract ClinicalTrial {
address public owner;
uint256 public trialStartTime;
uint256 public trialEndTime;
uint256 public enrollmentDeadline;
bool public trialComplete;


    struct Participant {
        bool enrolled;
        bool consentGiven;
        uint256 enrollmentTime;
    }
```

Smart Contract Use Cases - Clinical Trial

```
    mapping(address => Participant) public participants;
    event ParticipantEnrolled(address indexed participant);
    event ConsentGiven(address indexed participant);
    event TrialCompleted();
    modifier onlyOwner() {
        require(msg.sender == owner, "Only the owner can perform this action");
        _;
    }
    modifier isEnrolled() {
        require(participants[msg.sender].enrolled, "Participant is not enrolled");
        _;
    }
    modifier hasGivenConsent() {
        require(participants[msg.sender].consentGiven, "Participant has not given
consent");
        _;
    }
```

Smart Contract Use Cases - Clinical Trial

```
    constructor(uint256 _trialDurationDays, uint256 _enrollmentDeadlineDays) {
        owner = msg.sender;
        trialStartTime = block.timestamp;
        trialEndTime = trialStartTime + (_trialDurationDays * 1 days);
        enrollmentDeadline = trialStartTime + (_enrollmentDeadlineDays * 1 days);
        trialComplete = false;
    }

    function enroll() external {
        require(block.timestamp < enrollmentDeadline, "Enrollment period has
ended");
        require(!participants[msg.sender].enrolled, "Participant already enrolled");

        participants[msg.sender].enrolled = true;
        participants[msg.sender].enrollmentTime = block.timestamp;

        emit ParticipantEnrolled(msg.sender);
    }
```

Smart Contract Use Cases - Clinical Trial

```
    function giveConsent() external isEnrolled {
        require(!participants[msg.sender].consentGiven, "Consent already given");
        participants[msg.sender].consentGiven = true;

        emit ConsentGiven(msg.sender);
    }
    function completeTrial() external onlyOwner {
        require(block.timestamp >= trialEndTime, "Trial period has not ended yet");
        require(!trialComplete, "Trial has already been completed");
        trialComplete = true;


        emit TrialCompleted();


    }
    function getParticipantStatus(address _participant) external view returns (bool enrolled,
bool consentGiven, uint256 enrollmentTime) {
        enrolled = participants[_participant].enrolled;
        consentGiven = participants[_participant].consentGiven;
        enrollmentTime = participants[_participant].enrollmentTime;
    }
```

Smart Contract Use Cases - Clinical Trial

Contract Variables:

- owner: The address of the contract owner (presumably the entity conducting the clinical trial).
- trialStartTime: The timestamp when the clinical trial begins.
- trialEndTime: The timestamp when the clinical trial is expected to end.
- enrollmentDeadline: The timestamp indicating the deadline for participant enrollment.
- trialComplete: A boolean flag to indicate whether the trial has been completed or not. Participant Struct:
- Participant is a struct that holds information about each participant in the trial. It includes:
  - enrolled: A boolean flag indicating whether the participant is enrolled in the trial.
  - consentGiven: A boolean flag indicating whether the participant has given consent.
  - enrollmentTime: The timestamp when the participant enrolled in the trial.

Smart Contract Use Cases - Clinical Trial

Mapping:

- participants: This mapping associates Ethereum addresses with participant information. It allows you to look up participant details based on their Ethereum address.

Events:

- ParticipantEnrolled: An event emitted when a participant successfully enrolls in the trial.
- ConsentGiven: An event emitted when a participant gives their consent.
- TrialCompleted: An event emitted when the trial is marked as completed.

Modifiers:

- onlyOwner: A modifier that ensures that only the contract owner (presumably the clinical trial organizer) can execute certain functions.
- isEnrolled: A modifier that ensures that the caller is enrolled in the trial. hasGivenConsent: A modifier that ensures that the caller has given consent.

Constructor:

- When the contract is deployed, the constructor sets initial values for owner, trialStartTime, trialEndTime, enrollmentDeadline, and trialComplete. It takes parameters for the trial duration and enrollment deadline in days.

Smart Contract Use Cases - Clinical Trial

Enrollment Function:

- enroll(): Allows a participant to enroll in the trial if the enrollment period has not ended and the participant is not already enrolled. It sets the participant's enrollment status and records the enrollment time.

Consent Function:

- giveConsent(): Allows an enrolled participant to give consent for the trial. It checks that the participant is enrolled and has not already given consent.

Completion Function:

- completeTrial(): Allows the contract owner to mark the trial as completed if the trial period has ended and the trial has not already been marked as completed.

Participant Status Function:

- getParticipantStatus(): Allows anyone to query the enrollment status, consent status, and enrollment time of a participant by providing their Ethereum address.

THANK-YOU