Chapter 3: Ethereum Blockchain

By Maitri Hingu

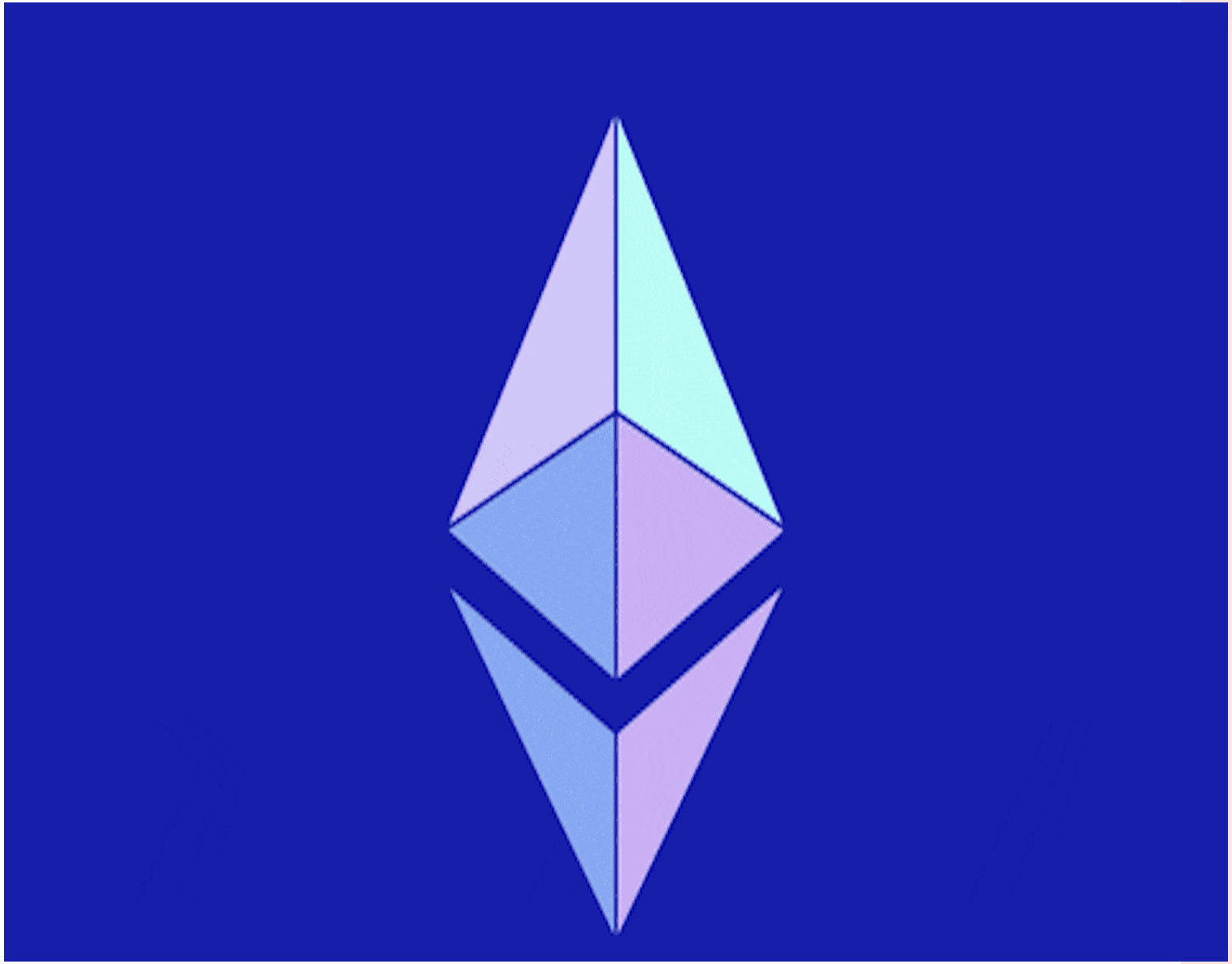
Agenda




- Overview of Ethereum, Network, and Structure
- Ether & Gas Points & Ether Mining
- Ethereum Operations & its Wallet
- Decentralized Autonomous Organization
- Proof of Stack in Ethereum
- Smart Contracts
- Creating Smart Contract Using Solidity

By Maitri Hingu

Overview of Ethereum, Network, and Structure



Overview of Ethereum, Network, and Structure

Ethereum is a decentralized, open-source blockchain platform that enables the creation and execution of smart contracts and decentralized applications (DApps). It was proposed by Vitalik Buterin in late 2013 and development began in early 2014, with the network officially launching on July 30, 2015. Ethereum is designed to be a global, tamper-resistant computing platform that allows developers to build and deploy decentralized applications.

Blockchain:

- Ethereum operates on a blockchain, which is a distributed and decentralized ledger that records all transactions across a network of computers.
- The Ethereum blockchain consists of a series of blocks, with each block containing a list of transactions. These blocks are linked together in a chronological chain, forming the entire transaction history of the network.

Overview of Ethereum, Network, and Structure

Smart Contracts:

- One of the defining features of Ethereum is its support for smart contracts. Smart contracts are self-executing contracts with the terms of the agreement directly written into code. They automatically enforce and execute the terms when predefined conditions are met.
- Solidity is the primary programming language used for developing smart contracts on the Ethereum platform.

Overview of Ethereum, Network, and Structure

Ether (ETH):

- Ether is the native cryptocurrency of the Ethereum network. It is used to compensate participants who perform computations and validate transactions (miners), as well as to pay for transaction fees and computational services within the network.
- Ether can also be used as a form of digital currency and is traded on various cryptocurrency exchanges.

Gas and Transaction Fees:

- Gas is a unit that measures the computational effort required to execute operations or run smart contracts on the Ethereum network.
- Users pay transaction fees in Ether (gas fees) to compensate miners or validators for the computational resources required to process and validate their transactions.

Overview of Ethereum, Network, and Structure

Gas Price:

- Gas price is the amount of Ether a user is willing to pay per unit of gas. Miners or validators prioritize transactions with higher gas prices because it increases their potential rewards.

Consensus Mechanism:

- Ethereum currently uses a Proof-of-Stake (PoS) consensus mechanism called Ethereum 2.0. However, it previously used Proof-of-Work (PoW) before transitioning to PoS to address scalability and environmental concerns.
- PoS relies on validators who lock up a certain amount of cryptocurrency as collateral to propose and validate new blocks. Validators are chosen to create new blocks based on the amount of cryptocurrency they hold and are willing to "stake."

Overview of Ethereum, Network, and Structure

Decentralized Applications (DApps):

- Ethereum allows developers to build decentralized applications on its platform. These applications are called DApps and operate on the Ethereum blockchain, utilizing smart contracts to execute logic and manage data.
- DApps can cover a wide range of functionalities, including finance, gaming, supply chain management, and more.

Ethereum Virtual Machine (EVM):

- The Ethereum Virtual Machine is a runtime environment that executes smart contracts on the Ethereum network. It is a crucial component that allows for the decentralized and trustless execution of code across the network.

Overview of Ethereum, Network, and Structure

Nodes:

- Nodes are individual computers that participate in the Ethereum network. There are different types of nodes, including full nodes that store the entire blockchain and validate transactions, and lightweight nodes that rely on full nodes for transaction verification.

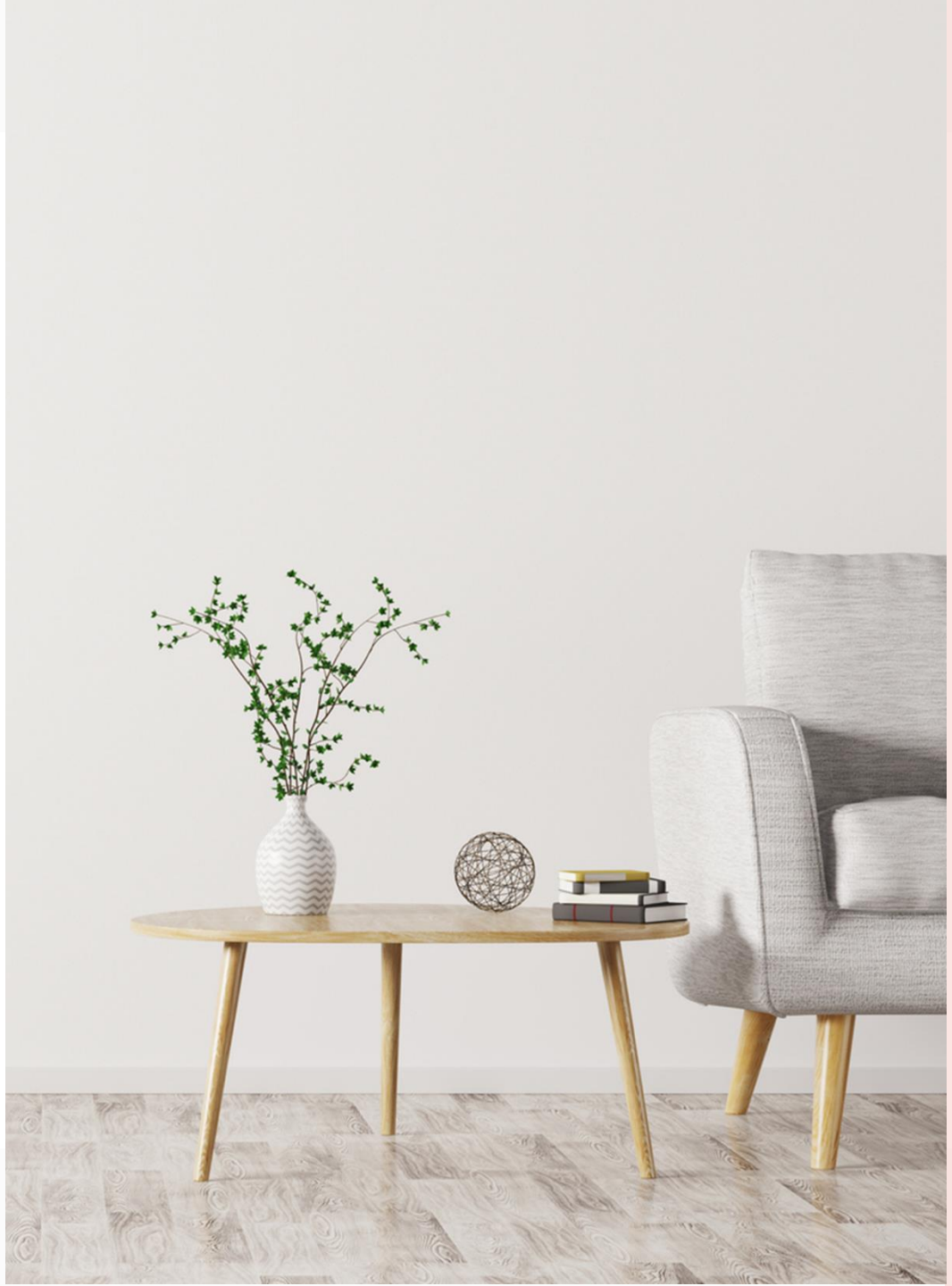
Hard Forks:

- Ethereum has undergone several hard forks to implement upgrades and improvements. Notable examples include the Byzantium and Constantinople upgrades. Hard forks require network-wide consensus to implement changes and can lead to the creation of separate chains (e.g., Ethereum and Ethereum Classic).

Overview of Ethereum, Network, and Structure

In summary, Ethereum is a decentralized blockchain platform that facilitates the creation and execution of smart contracts and decentralized applications. It features a native cryptocurrency (Ether), a consensus mechanism (Proof-of-Stake), and a vibrant ecosystem of developers building various decentralized applications on top of the Ethereum blockchain.

Structure of Ethereum



Structure of Ethereum

The internal structure of Ethereum involves various components that work together to create a decentralized, programmable blockchain platform.

Accounts:

- Ethereum has two types of accounts: Externally Owned Accounts (EOAs) and Contract Accounts.
- EOAs are controlled by private keys and are used by individuals to send and receive Ether (ETH) or initiate transactions on the network.
- Contract Accounts store and execute smart contracts. They are controlled by the code of the smart contract and managed by transactions sent from EOAs.

Structure of Ethereum

Externally Owned Accounts (EOAs) on Ethereum:

- EOAs are like individual treasure chests controlled by private keys.
- They have direct control over their funds and transactions.
- Each EOA has its own unique private key for access.
- EOAs can send Ether (ETH) and interact with smart contracts on the Ethereum blockchain.
- These are used for receiving mining rewards and transaction fees. Miners typically use EOAs to receive the rewards earned from mining new blocks and the fees collected from transactions included in those blocks.

Structure of Ethereum

Contract Accounts (CAs) on Ethereum:

- Unlike Externally Owned Accounts (EOAs) with private keys, Contract Accounts don't have traditional wallets.
- They're special accounts for executing smart contracts, not owned by individuals.
- Contract Accounts hold smart contract code and data, executed by the Ethereum Virtual Machine (EVM).
- Funds held by Contract Accounts are managed by smart contract logic, not private keys.
- Miners may also have Contract Accounts, which are used for deploying smart contracts or interacting with decentralized applications (DApps). While EOAs are primarily used for receiving rewards, CAs allow miners to engage in more complex interactions within the Ethereum ecosystem.

Structure of Ethereum

State:

- The state of Ethereum represents the current state of all accounts and smart contracts on the network. It includes account balances, contract storage, and the code of deployed smart contracts.
- Changes in the state occur as a result of transactions and smart contract executions.

Transactions:

- Transactions are messages sent by externally owned accounts to the Ethereum network to perform various actions, such as sending Ether, interacting with smart contracts, or deploying new contracts.
- Each transaction includes details like the sender, recipient, amount of Ether, and optional data.

Structure of Ethereum

Blocks:

- Transactions are grouped into blocks, which are then added to the Ethereum blockchain. Blocks contain a header and a list of transactions.
- The header includes metadata like the block number, timestamp, and a reference to the previous block (creating a chain of blocks).

Ethereum Virtual Machine (EVM):

- The EVM is a crucial component of Ethereum's internal structure. It is a Turing-complete virtual machine that executes bytecode of smart contracts.
- Every node in the Ethereum network runs an instance of the EVM to validate and execute transactions and smart contracts.

Structure of Ethereum

Gas:

- Gas is a unit of measure for the computational work performed on the Ethereum network. It represents the cost of running operations in the EVM.
- Users pay for gas using Ether when they initiate transactions or execute smart contracts. The gas limit is the maximum amount of gas a user is willing to spend on a transaction.

Gas Price:

- Gas price is the amount of Ether a user is willing to pay per unit of gas. Miners or validators prioritize transactions with higher gas prices because it increases their potential rewards.

Structure of Ethereum

Mining/Validation:

- In the Proof-of-Work (PoW) era, miners competed to solve cryptographic puzzles to validate transactions and add blocks to the blockchain. In Proof-of-Stake (PoS), validators are chosen to propose and validate blocks based on the amount of cryptocurrency they hold and are willing to stake.

Consensus Mechanism:

- Ethereum's consensus mechanism ensures that all nodes agree on the state of the blockchain. PoW and PoS are examples of consensus mechanisms used in Ethereum at different stages of its development.

Forks:

- Ethereum has undergone hard forks to implement upgrades and improvements. Forks can be planned upgrades or contentious events that result in a split, creating separate chains (e.g., Ethereum and Ethereum Classic).

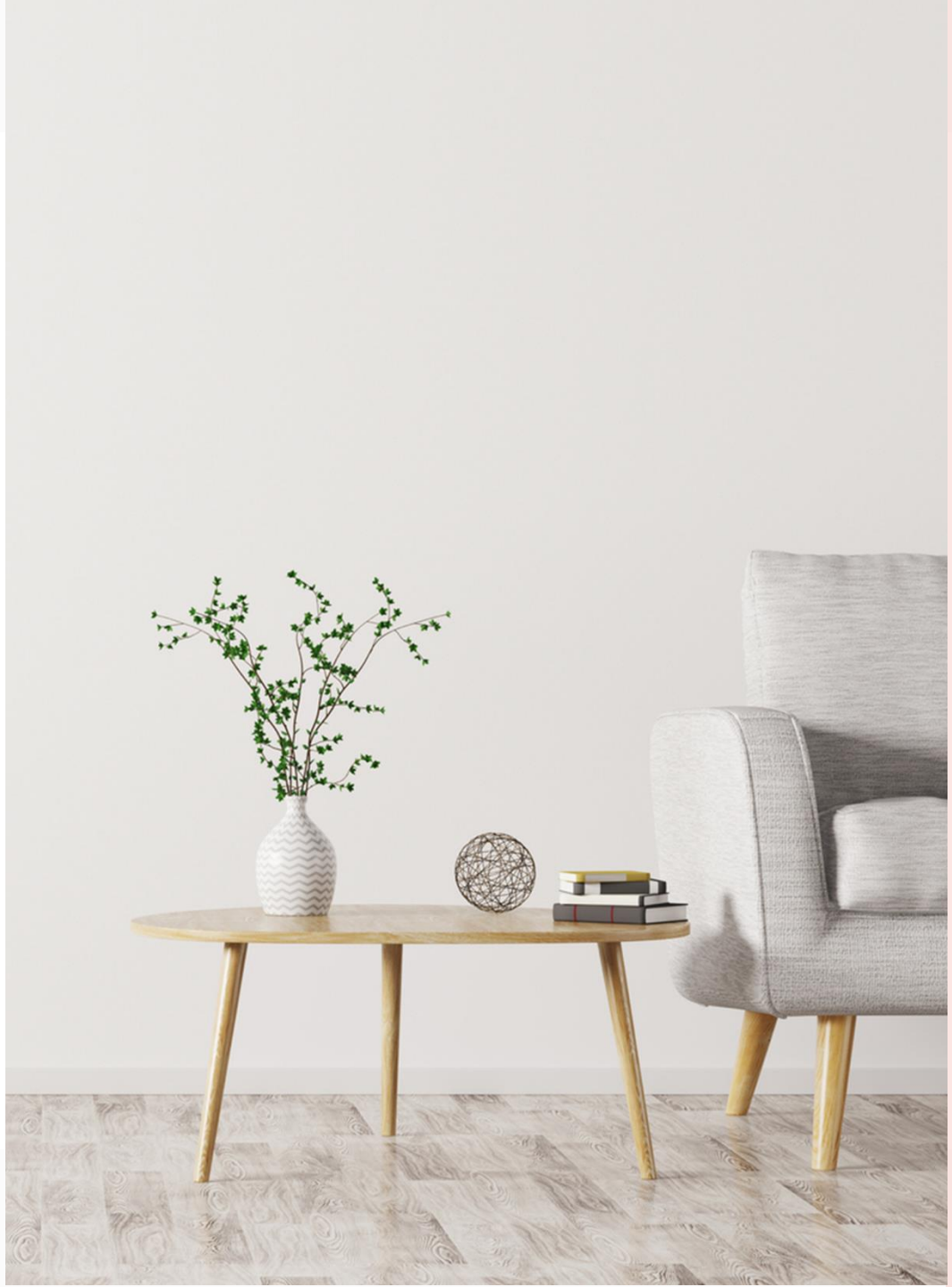
Structure of Ethereum

Network Nodes:

- Nodes are individual computers that participate in the Ethereum network. Full nodes store the entire blockchain, validate transactions, and relay information to other nodes, contributing to the network's security and decentralization.

Understanding the internal structure of Ethereum involves grasping the roles played by accounts, state, transactions, blocks, the EVM, gas, mining or validation processes, consensus mechanisms, forks, and the network nodes. Together, these elements create a decentralized and programmable platform that supports smart contracts and decentralized applications.

Ethereum Operations & its Wallet



Ethereum Operations & its Wallet

Ethereum Operations:

- **Sending and receiving Ether:** Users can send and receive Ether (ETH), the native cryptocurrency of the Ethereum blockchain, to and from other addresses.
- **Interacting with smart contracts:** Users can execute predefined actions within smart contracts, such as token transfers, decentralized finance (DeFi) transactions, and voting.
- **Token transfers:** Ethereum supports the creation and transfer of tokens through smart contracts, enabling users to exchange assets, collectibles, or participation rights.
- **Contract deployments:** Users can deploy their own smart contracts onto the Ethereum blockchain to implement custom functionalities and decentralized applications.

Ethereum Operations & its Wallet

Ethereum Wallets:

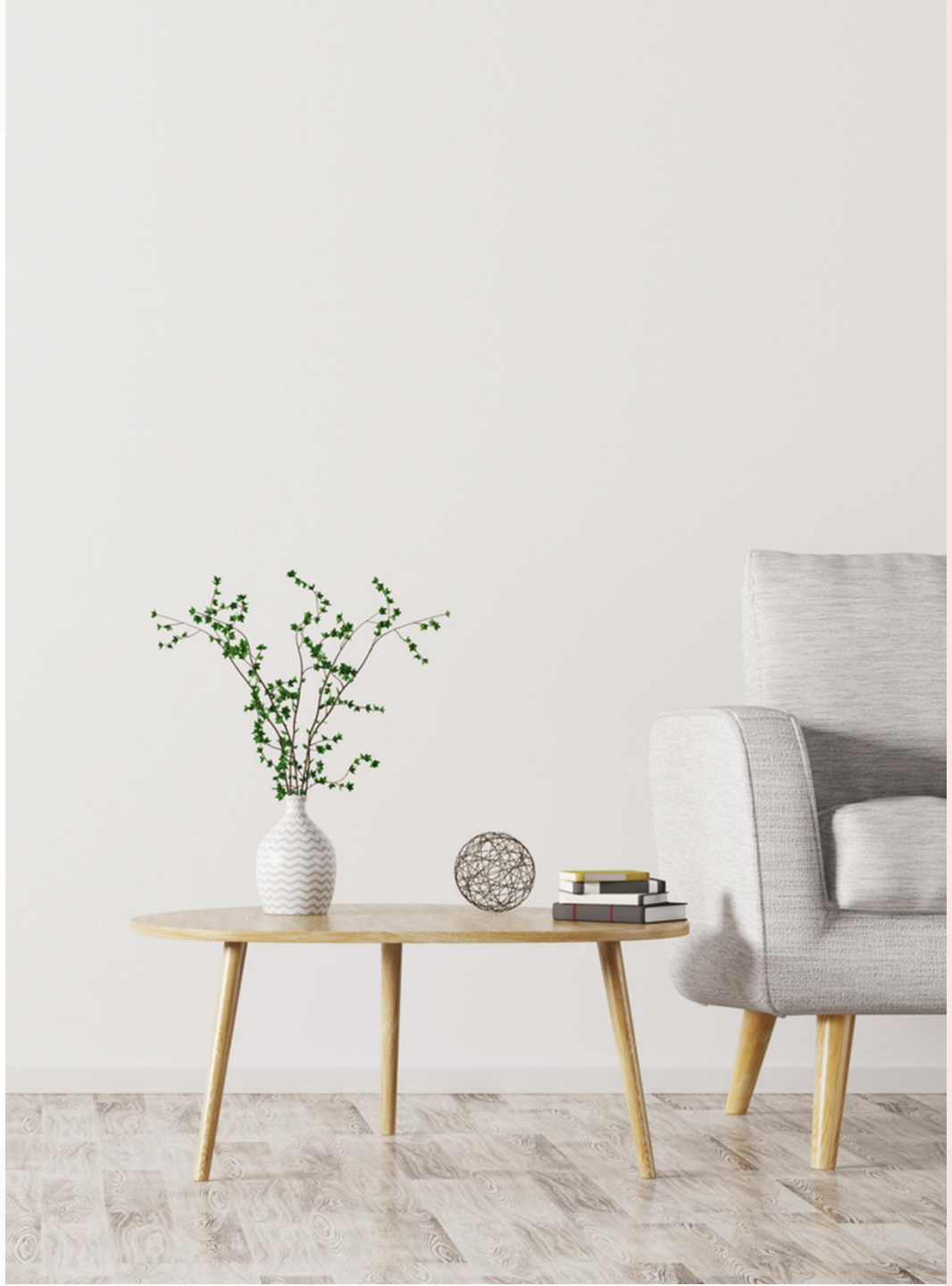
- **Software wallets (desktop, mobile, web):** Applications or software programs that allow users to store, manage, and interact with Ether and tokens on various devices.
- **Hardware wallets:** Physical devices designed for securely storing cryptocurrency keys offline, providing enhanced security compared to software wallets. (Example: Ledger Nano S)
- **Paper wallets:** Offline storage method involving printing private keys and addresses on paper, offering an extra layer of security. (Example: MyEtherWallet (MEW))
- **Custodial wallets:** Wallets provided by third-party services, such as exchanges or online wallet providers, where users deposit funds into accounts managed by the service provider.

Ethereum Operations & its Wallet

Wallet Functionality:

- **Generating and managing Ethereum addresses:** Wallets create and manage Ethereum addresses for sending and receiving transactions.
- **Viewing transaction history:** Users can track their transaction history to monitor incoming and outgoing payments.
- **Sending and receiving Ether and tokens:** Wallets enable users to send and receive Ether and Ethereum-based tokens to and from other addresses.
- **Interacting with smart contracts:** Some wallets allow users to directly interact with smart contracts, facilitating participation in decentralized applications and token transfers.
- **Security features:** Wallets offer security features such as password protection, encryption, multi-factor authentication, and backup options to safeguard users' funds and private keys.

DAO (Decentralized Autonomous Organization)



DAO (Decentralized Autonomous Organization)

The DAO (Decentralized Autonomous Organization) was a groundbreaking project on the Ethereum blockchain designed to operate as a decentralized venture capital fund.

Creation of The DAO: The DAO was created through a crowdfunding campaign in 2016. Investors sent Ether (ETH) to The DAO's smart contract address in exchange for DAO tokens, which represented their ownership stake in the organization.

Proposal Submission: Anyone could submit a proposal to The DAO outlining a project or idea they wanted to fund. Proposals typically included details such as the project's goals, budget, and timeline.

Voting: DAO token holders had the power to vote on proposed projects using their tokens. Each token equated to one vote. The voting process was transparent and executed directly on the Ethereum blockchain.

DAO (Decentralized Autonomous Organization)

Funding: If a proposal received enough votes and passed the voting threshold, funds from The DAO's treasury were automatically released to the project's smart contract address. This provided funding for the proposed project to move forward.

Project Execution: Once funded, the project team could access the Ether and begin executing their proposal. Progress and expenditures were transparently recorded on the Ethereum blockchain, allowing DAO token holders to monitor the project's development.

Rewards and Returns: If a funded project generated profits or revenue, these returns would flow back to The DAO's treasury. DAO token holders could then vote on how to allocate these returns, whether reinvesting them into new projects, distributing them to token holders as dividends, or other options.

DAO (Decentralized Autonomous Organization)

The DAO aimed to revolutionize venture capital by enabling decentralized decision-making and investment. However, it faced challenges, particularly with a vulnerability in its smart contract code that led to a significant exploit. This exploit resulted in a contentious hard fork of the Ethereum blockchain and the creation of Ethereum (ETH) and Ethereum Classic (ETC) as separate chains.

While The DAO experiment ultimately encountered difficulties, it provided valuable insights into decentralized governance, smart contract security, and the potential of blockchain technology for collective decision-making and investment.

DAO Attack

The DAO attack occurred on June 17, 2016, when a vulnerability in the code of The DAO smart contract was exploited, leading to the theft of approximately 3.6 million Ether.

The solution to address the DAO attack was a contentious one within the Ethereum community. Initially, there was a proposal for a soft fork, which aimed to freeze the stolen Ether and prevent the attacker from accessing the funds. However, this proposal faced significant debate and opposition due to concerns about the implications for blockchain immutability and decentralization.

Ultimately, the Ethereum community decided to implement a hard fork to reverse the effects of the exploit. The hard fork, known as the DAO Hard Fork or DAO bailout, involved a modification to the Ethereum protocol that effectively rolled back the blockchain to a state before the DAO exploit occurred.

DAO Attack

This decision led to the creation of two separate blockchains: Ethereum (ETH) and Ethereum Classic (ETC), with the majority of the community supporting the hard-forked Ethereum (ETH) chain. The hard fork successfully recovered the stolen Ether and mitigated the impact of the exploit.

After DAO Hard Fork: Ethereum, Ethereum Classic

Imagine you have a big treasure chest in your backyard, and all your friends can put their coins into the chest to build something cool together, like a giant playground. This treasure chest is like a special computer program called "The DAO" on the internet.

Now, one day, someone found a tricky way to sneak into the treasure chest and take out lots of coins. They did this by using a special trick in the computer program that made it give out more coins than it was supposed to.

This sneaky person didn't just take a few coins; they took a whole bunch – millions of them! This made everyone very sad because there weren't enough coins left to build the awesome playground that everyone was excited about.

To fix the problem, the friends had a big meeting and talked about what they could do. Some friends suggested they could use a magic spell called a "soft fork" to freeze the treasure chest and stop the thief from spending the stolen coins. But not everyone agreed on using the magic spell because it would change how things work.

By Maitri Hingu

After DAO Hard Fork: Ethereum, Ethereum Classic

So, instead of the magic spell, they decided to use an even bigger magic spell called a "hard fork." This spell would rewind time, like going back in a storybook, to a time before the sneaky person took the coins. This way, they could start fresh and build their amazing playground without any problems.

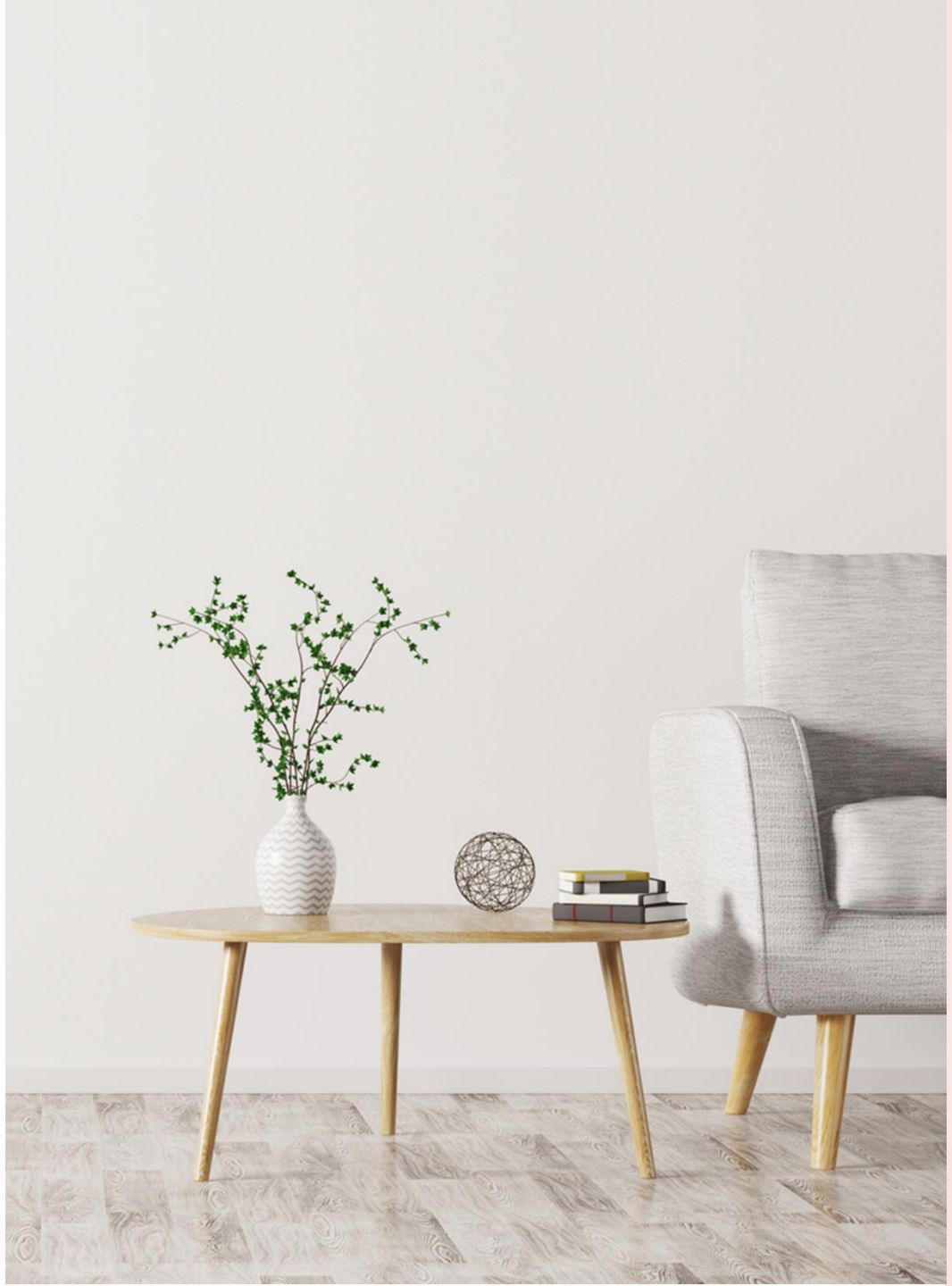
But when they used the big magic spell, something interesting happened. The friends split into two groups – one group continued with the new story where the coins were safe (Ethereum), and the other group wanted to keep the old story as it was, even with the stolen coins (Ethereum Classic).

So, in the end, there were two different stories – one where they rewound time to fix the problem (Ethereum), and another where they kept going with the old story (Ethereum Classic).

And that's how a tricky person tried to take away the coins from the treasure chest, and the friends had to use magical spells to make things right again!

By Maitri Hingu

Proof of Stack in Ethereum



Proof of Stack in Ethereum

Proof of Stake (PoS) is a consensus mechanism used in Ethereum to validate and add new blocks to the blockchain.

Validator Selection:

- Ethereum selects validators to participate in block validation based on their stake, i.e., the amount of cryptocurrency they hold and commit as collateral.
- Validators are chosen through a deterministic algorithm that considers factors such as the amount of cryptocurrency staked and the validator's reputation.

Block Proposal:

- Validators take turns proposing new blocks to be added to the blockchain.
- The probability of being selected to propose a block is directly proportional to the validator's stake.

Proof of Stack in Ethereum

Block Validation:

- Validators validate the proposed block by verifying transactions and ensuring they adhere to the protocol rules.
- Validators also check for any invalid transactions, double spending attempts, or other malicious activities.

Commitment and Consensus:

- Validators commit their stake as collateral to vouch for the validity of the proposed block.
- Consensus is reached when a supermajority of validators (e.g., two-thirds) agree on the validity of the proposed block.
- If consensus is reached, the proposed block is added to the blockchain, and validators are rewarded for their participation.

Proof of Stack in Ethereum

Incentivization and Penalties:

- Validators are incentivized to act honestly and follow the rules of the network.
- Validators earn rewards for successfully proposing and validating blocks.
- Validators may face penalties, such as losing a portion of their stake, for malicious behavior or attempting to manipulate the consensus process.

Finality and Security:

- Blocks added to the blockchain through Proof of Stake are considered final once they are confirmed by a sufficient number of validators.
- The security of the network relies on the economic incentives of validators to act honestly, as validators have a financial stake in the network's integrity.

Proof of Stack in Ethereum

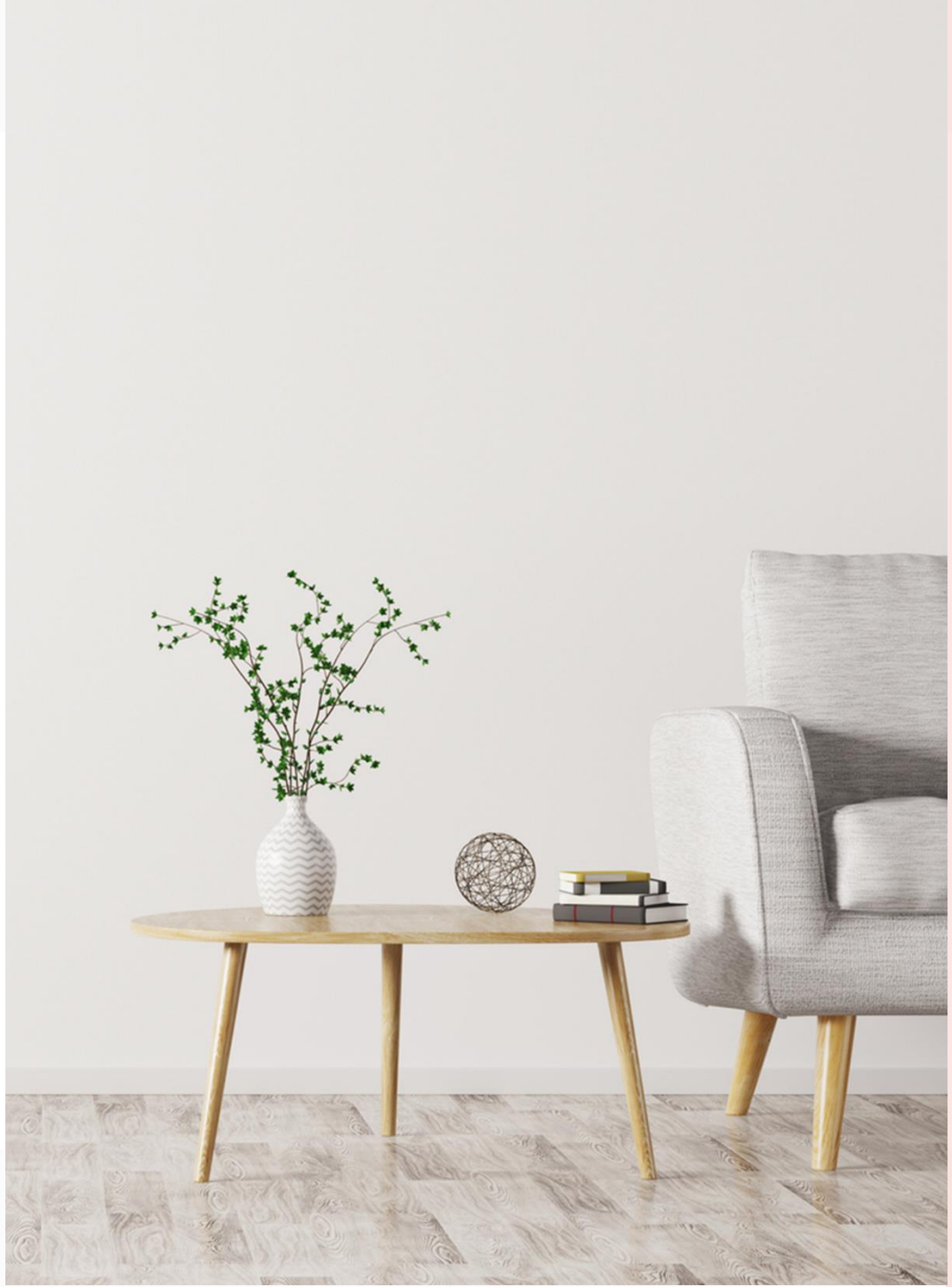
Dynamic Participation:

- Validators can join or leave the network at any time by staking or unstaking their cryptocurrency.
- The network adjusts the selection of validators dynamically based on changes in their stake and participation levels.

Epochs and Slashing:

- The PoS protocol operates in epochs, which are fixed time intervals during which validators participate in block validation.
- Validators may face slashing penalties for violating protocol rules, such as signing conflicting blocks or attempting to compromise the network's security.

Smart Contracts



Smart Contract

A smart contract is a self-executing digital contract with predefined rules and conditions written in code. It automatically enforces and executes the terms of the agreement when specific conditions are met, eliminating the need for intermediaries and providing trustless and decentralized automation of transactions and agreements.

Creation:

- A developer writes a smart contract using the Solidity programming language or other compatible languages.
- The smart contract code defines the contract's functionality, including its variables, functions, and logic.

Smart Contract

Compilation:

- The smart contract code is compiled into bytecode, which is the low-level representation of the contract's instructions.
- The compiled bytecode is then deployed to the Ethereum blockchain.

Deployment:

- A user deploys the compiled smart contract bytecode to the Ethereum network by sending a deployment transaction.
- This transaction includes the bytecode of the contract and any necessary initialization parameters.
- Miners on the Ethereum network validate the deployment transaction and add it to a block, effectively deploying the smart contract to the blockchain.

Smart Contract

Instantiation:

- Once the deployment transaction is confirmed and included in a block, the smart contract is instantiated, and a unique address is assigned to it on the Ethereum blockchain.
- This address serves as the identifier for interacting with the deployed smart contract.

Interactions:

- Users interact with the deployed smart contract by sending transactions to its address.
- These transactions invoke specific functions defined in the smart contract code and can include data or parameters required for the function execution.
- Each interaction with the smart contract is recorded on the blockchain as a transaction, ensuring transparency and immutability.

Smart Contract

Execution:

- Miners validate and execute the transactions sent to the smart contract by executing the corresponding functions defined in its code.
- The smart contract's code is executed on the Ethereum Virtual Machine (EVM), a decentralized runtime environment that ensures consistency and determinism across all nodes on the network.

State Changes:

- The execution of transactions may result in changes to the state of the smart contract, such as updating its variables, modifying its storage, or emitting events.
- These state changes are recorded on the Ethereum blockchain, reflecting the current state of the smart contract after each transaction execution.

Smart Contract

Validation and Consensus:

- Miners on the Ethereum network validate and reach consensus on the transactions and state changes associated with the smart contract.
- Consensus mechanisms, such as Proof of Work (PoW) or Proof of Stake (PoS), ensure the integrity and security of the blockchain and prevent double-spending or fraud.

Transaction Fees:

- Users pay transaction fees, known as gas fees, for interacting with smart contracts on the Ethereum network.
- Gas fees compensate miners for their computational resources and incentivize them to process transactions efficiently.

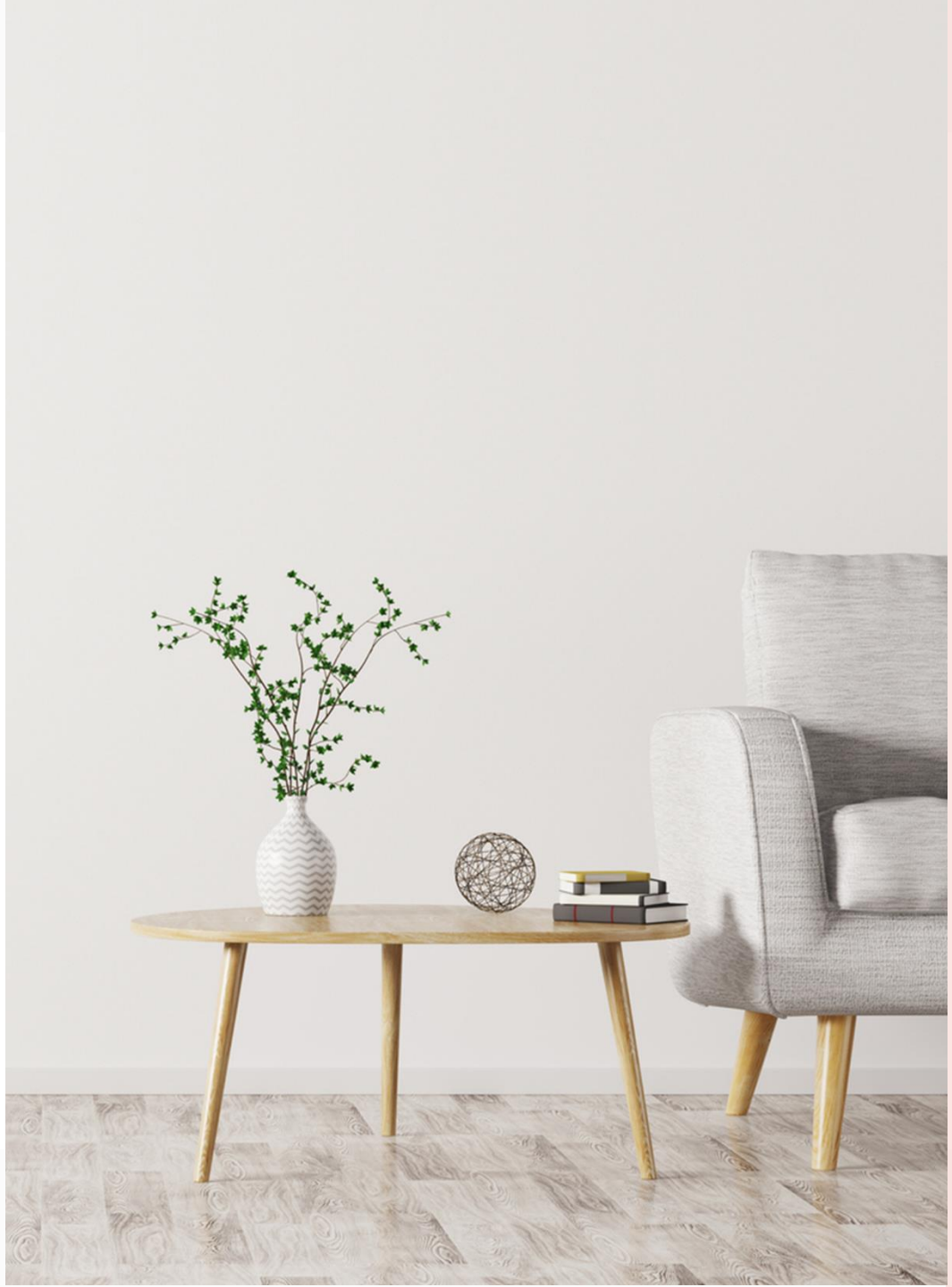
Smart Contract

Finality and Immutability:

- Once transactions involving the smart contract are confirmed and included in a block, they are considered final and immutable.
- The smart contract's code and state are stored on the blockchain indefinitely, providing transparency, auditability, and tamper resistance.

In summary, smart contracts in Ethereum are self-executing contracts with predefined rules and logic deployed on the blockchain. They enable decentralized and trustless interactions, automate processes, and facilitate a wide range of applications, including decentralized finance (DeFi), digital identity, supply chain management, and more.

Solidity Language for Smart Contracts



Solidity Language for Smart Contracts

Solidity is a high-level, statically-typed programming language used for writing smart contracts on various blockchain platforms, with Ethereum being the most prominent. Developed specifically for Ethereum, Solidity enables developers to define the logic of decentralized applications (DApps) and smart contracts that run on the Ethereum Virtual Machine (EVM).

- **Purpose:** Solidity is designed to enable the creation of smart contracts, which are self-executing contracts with the terms of the agreement directly written into code. These contracts automatically enforce and execute the terms of the agreement when predetermined conditions are met.
- **Syntax:** Solidity syntax is similar to that of JavaScript and is influenced by languages like C++ and Python. It supports features such as inheritance, libraries, and complex user-defined types.

Solidity Language for Smart Contracts

- **Smart Contract Development:** Solidity facilitates the development of smart contracts by providing constructs for state variables, functions, events, modifiers, and more. Developers can define the behavior of their smart contracts using Solidity's expressive syntax.
- **Security:** Solidity incorporates security features and best practices to help developers write secure smart contracts. However, writing secure contracts still requires careful consideration of potential vulnerabilities and adherence to best practices.
- **Compatibility:** Solidity is primarily used for Ethereum smart contracts but can potentially be used on other blockchain platforms that support the EVM or have compatible virtual machines.

Solidity Language for Smart Contracts

- **Community and Ecosystem:** Solidity has a vibrant community of developers and contributors who actively maintain the language and its associated tools. It also has a rich ecosystem of libraries, frameworks, and development tools to aid in smart contract development.
- **Versioning:** Solidity undergoes regular updates and improvements. Developers should specify the version of Solidity they are using in their contracts to ensure compatibility and to take advantage of the latest features and optimizations.

Overall, Solidity plays a crucial role in the Ethereum ecosystem by enabling the creation of decentralized applications and facilitating the execution of smart contracts, which form the backbone of various blockchain-based use cases such as decentralized finance (DeFi), non-fungible tokens (NFTs), decentralized autonomous organizations (DAOs), and more.

Solidity Language for Smart Contracts

Let's Learn Solidity. 😎

- **Comments:**

  Single-line comments are denoted by //.

  Multi-line comments are enclosed within /* */.

  Example:

```
// This is a single-line comment


/*
  This is a
  multi-line comment
*/
```

Solidity Language for Smart Contracts

- **Pragma Directive:**

  The pragma directive is used to specify the compiler version to be used for compiling the contract.

  It's recommended to specify a pragma directive to prevent compatibility issues with future compiler versions.

  Example:

  pragma solidity ^0.8.0;

  pragma solidity >=0.6.12 <0.9.0;

Solidity Language for Smart Contracts

- **Pragma Directive:**

  pragma solidity ^0.8.0;

  This directive specifies that the contract should be compiled using a Solidity compiler version equal to or greater than 0.8.0, but less than 0.9.0.

  The caret (^) symbol is a caret-range operator. It means that the contract is compatible with any compiler version from 0.8.0 up to, but not including, 0.9.0.

  Using the caret-range operator allows for flexibility in specifying compatible compiler versions while ensuring that breaking changes introduced in major versions are not included.

Solidity Language for Smart Contracts

- **Pragma Directive:**

  pragma solidity >=0.6.12 <0.9.0;

  This directive specifies that the contract should be compiled using a Solidity compiler version equal to or greater than 0.6.12, but less than 0.9.0.

  The >= and < operators are used to define a range of compatible compiler versions.

  The >= operator indicates that the contract is compatible with compiler versions equal to or greater than 0.6.12.

  The < operator indicates that the contract is compatible with compiler versions less than 0.9.0.

  Together, these operators define a range of compatible compiler versions between 0.6.12 (inclusive) and 0.9.0 (exclusive).

Solidity Language for Smart Contracts

- **Pragma Directive:**

  In summary, the differences between the operators used in pragma solidity directives are as follows:

  ^: Denotes a caret-range operator. Specifies a compatible range of compiler versions, allowing for flexibility in selecting compatible versions while excluding major breaking changes.

  >= and <: Denote operators for defining a range of compatible compiler versions. Specifies a range of versions between the specified minimum (inclusive) and maximum (exclusive) versions.

Solidity Language for Smart Contracts

- **Contract Declaration:**

  A contract in Solidity is similar to a class in object-oriented programming languages.

  It encapsulates data (state variables) and functions that can interact with this data.

  Example:

```
contract MyContract {
    // State variables and functions will be declared here
}
```

Solidity Language for Smart Contracts

- **State Variables:**

  State variables represent the data stored permanently in the contract's storage.

  They can be of various types such as uint, int, bool, address, string, mapping, struct, etc.

  Example:

```
contract MyContract {
    uint public myNumber;          // Unsigned integer variable
    address public owner;          // Ethereum address variable
    string public myString;        // String variable
    bool public myBool;            // Boolean variable
}
```

Solidity Language for Smart Contracts

- **Visibility Specifiers:**

  Functions and state variables can have visibility specifiers to control access to them. The visibility specifiers include public, private, internal, and external.

Visibility Specifier	Description	Accessible From
Public	Accessible from inside and outside the contract. Solidity generates a getter function.	Inside the contract and externally via the generated getter function.
Private	Accessible only from within the contract where it is declared.	Only within the contract where it is declared.
Internal	Accessible from within the current contract and contracts that inherit from it.	Within the current contract and contracts that inherit from it.
External	Accessible only from outside the contract, typically by other contracts or external accounts.	Only externally, from outside the contract.

Solidity Language for Smart Contracts

- **Example of Visibility Specifiers:**

```
contract MyContract {
    uint public myNumber;          // Public state variable
    uint private myPrivateNumber;          // Private state variable
    function setMyPrivateNumber(uint _num) external {
        myPrivateNumber = _num;
    }
    function getMyPrivateNumber() external view returns (uint) {
        return myPrivateNumber;
    }
}
```

Solidity Language for Smart Contracts

- **Functions:**

  Functions in Solidity define the behavior of the contract.

  They can be of two types: external and internal.

  External functions can be called from outside the contract, while internal functions can only be called from within the contract.

Solidity Language for Smart Contracts

- **Example of Functions:**

```
contract MyContract {
    uint public myNumber;
    function setMyNumber(uint _num) external {
        myNumber = _num;
    }
        function getMyNumber() external view returns (uint) {
        return myNumber;
    }
}
```

Solidity Language for Smart Contracts

- **Constructor:**

  A constructor is a special function that gets executed only once when the contract is deployed.

  It's used to initialize contract state variables or perform any setup tasks.

Solidity Language for Smart Contracts

- **Example of Constructor:**

```
contract MyContract {
    uint public myNumber;
    // Constructor to initialize myNumber
    constructor(uint _initialNumber) {
        myNumber = _initialNumber;
    }
}
```

Thank You

Maitri Hingu

mkhingu@vnsgu.ac.in