SMART CONTRACTS

MODULE 3

Overview

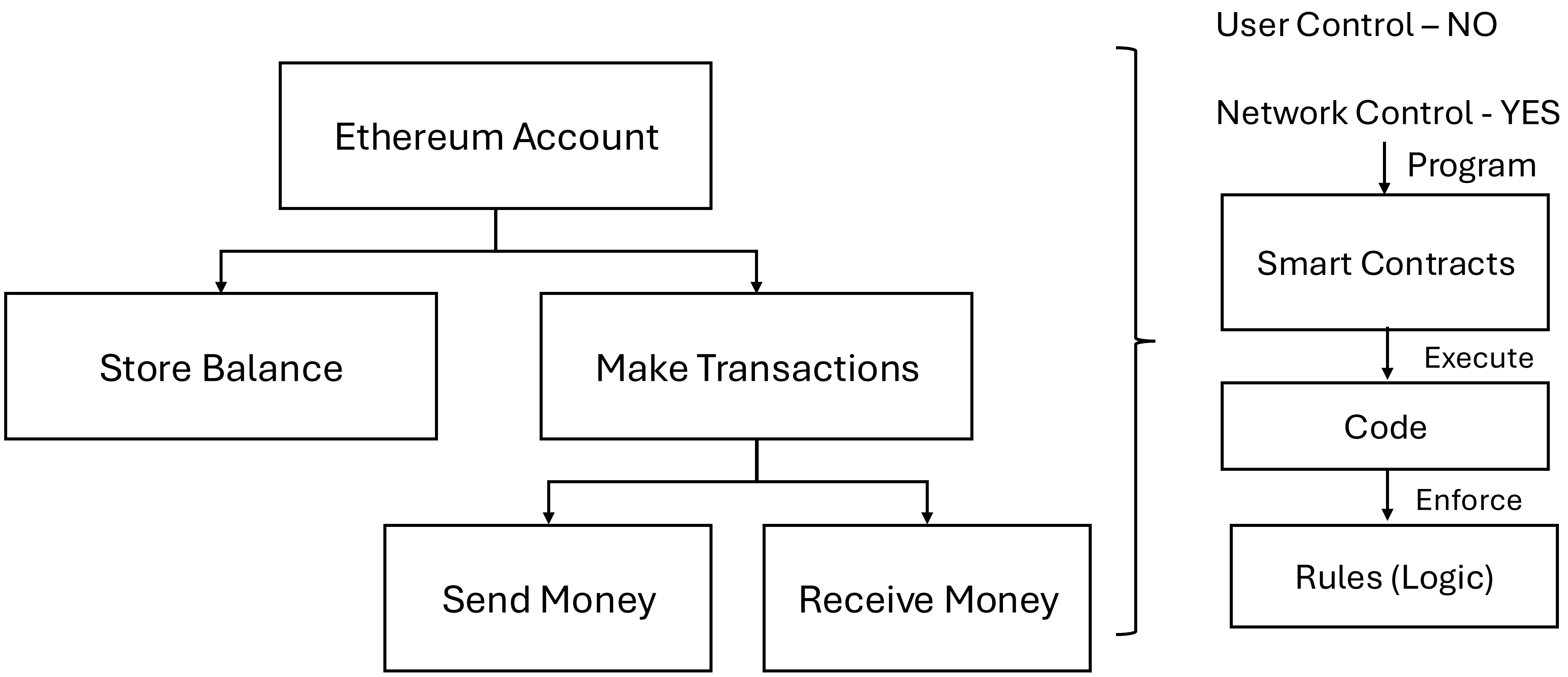
- Anatomy of a Smart Contracts
- Life Cycle
- Usage Patterns
- DLT-based smart contracts
- Use Cases
    - Healthcare Industry
    - Property Transfer

Smart Contracts

- A "smart contract" is simply a program that runs on the Ethereum blockchain.
- It is a collection of **code (its functions)** and **data (its state)** that **resides at a specific address on the Ethereum blockchain**.
- Written using Solidity
- Code → function
- Data → State

Smart Contracts

- Consider → Smart Contracts → Ethereum Account



Smart Contracts

- Smart Contract → Vending Machine
- Rules (Logic):
  - Money + Snack Selection = Snack dispensed
  - No Money + Snack Selection = No Snack dispensed
  - Money + No Snack Selection = No Snack dispensed

Smart Contracts

```
pragma solidity ^0.8.0;

contract VendingMachine {
  address public owner;
  uint public snackPrice = 1 ether;
  mapping(string => uint) public snackInventory;

  constructor() {
    owner = msg.sender;
    // Preload some snacks
    snackInventory["chips"] = 10;
    snackInventory["soda"] = 10;
    snackInventory["candy"] = 10;
  }

// Function to buy snack
  function buySnack(string memory snackName) public payable returns (string memory) {
    require(msg.value == snackPrice, "Incorrect amount sent!");
    require(snackInventory[snackName] > 0, "Snack not available!");

    // Dispense snack
    snackInventory[snackName] -= 1;
    return "Snack dispensed!";
  }

  // Only owner can restock
  function restockSnack(string memory snackName, uint amount) public {
    require(msg.sender == owner, "Only owner can restock!");
    snackInventory[snackName] += amount;
  }

  // Check balance
  function checkBalance() public view returns (uint) {
    return address(this).balance;
  }
}
```

Smart Contracts

- Permissionless → Anyone can write Smart Contracts
- Requirements:
- Smart Contact language
  - Solidity
  - Vyper
- Deployment of contract
  - ETH (Gas)
- Ethereum Virtual Machine
  - Compiling
  - Interpret (Bytecode)
  - Store

Total Gas = Amount of work done × Price per unit of gas

Example

Suppose you call a function to **mint an NFT**.

•Gas used = 100,000 units

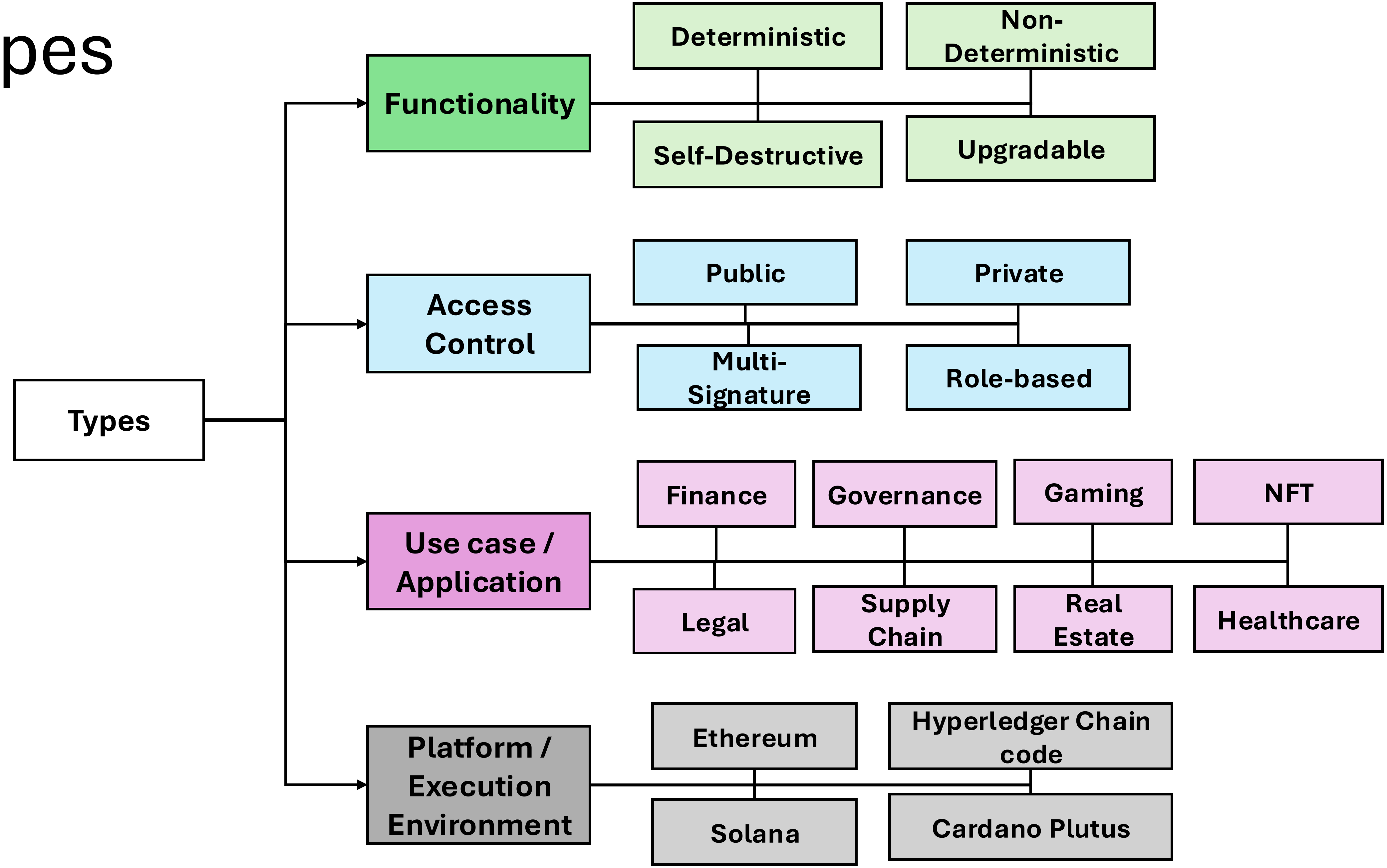
•Gas price = 30 Gwei (1 Gwei = 10^{-9} ETH)

💰 **Total Gas Fee** = 100,000 × 30 Gwei = 0.003 ETH

Features of Smart Contracts

Feature	Description	Example
1. Self-executing	Executes automatically when predefined conditions are met	Auto-transfer of funds when loan is repaid
2. Immutable	Once deployed, the contract code **cannot be changed** (unless a proxy pattern is used)	Prevents tampering or logic manipulation
3. Trustless	No need for third-party intermediaries	Buyer and seller interact directly
4. Deterministic	Produces the same result for the same input every time	E.g., f(x) = x * 2 always gives the same result
5. Transparent	Code and transactions are **visible** on the blockchain	Anyone can audit logic and behaviour
6. Tamper-proof	Data is cryptographically secured and cannot be altered	Voting data remains secure
7. Distributed execution	Runs on multiple nodes, ensuring **high availability and fault tolerance**	Contract continues even if some nodes fail
8. Cost-efficient	Reduces need for paperwork and middlemen	Automated insurance claims
9. Conditional logic	Uses if...else, loops, and conditions to perform complex actions	"If payment received, then ship product"
10. Event-triggered	Can emit **events/logs** based on on-chain actions	Logs transactions or user actions
11. Composable	Can **call other smart contracts** and interact with DeFi protocols or services	NFT marketplace calling token contracts
12. Version-controlled	Contract versions are linked through upgradeable proxies (optional)	Can upgrade logic without losing state

Types


Types
Functionality
Deterministic
Non-Deterministic
Self-Destructive
Upgradable
Access Control
Public
Private
Multi-Signature
Role-based
Use case / Application
Finance
Governance
Gaming
NFT
Legal
Supply Chain
Real Estate
Healthcare
Platform / Execution Environment
Ethereum
Hyperledger Chain code
Solana
Cardano Plutus

Based on Functionality

Type	Description	Example
Deterministic Contracts	Produces the same output for the same input every time (no external data)	Token transfer, voting systems
Non-deterministic Contracts	Relies on external data sources (oracles)	Weather-based insurance payouts
Self-destructible Contracts	Can be destroyed and removed from the blockchain	Temporary crowdfunding contracts
Upgradable Contracts	Allow logic upgrades without changing the contract address (via proxy pattern)	DeFi protocols like Aave

Based on Access Control

Type	Description	Example
Public Contracts	Any user can interact with the contract	ERC20 token contract
Private/Permissioned Contracts	Restricted access to specific roles or users	Enterprise contracts (Hyperledger)
Multi-signature Contracts	Requires multiple parties to approve transactions	Joint fund management
Role-based Contracts	Uses modifiers (e.g., onlyOwner, admin) to restrict certain actions	Admin-controlled NFT minting

Based on Use Case / Application

Type	Description	Example
Financial Smart Contracts	Handles money: payments, loans, escrows, yield farming	Uniswap, Compound
Governance Contracts	Enables on-chain voting and decision-making	DAO voting contracts
Supply Chain Contracts	Tracks product origin, status, and events	IBM Food Trust
Legal/Identity Contracts	Manages digital identity, credentials, legal agreements	Self-sovereign identity (SSI)
Gaming Contracts	Enables asset ownership, in-game logic	Axie Infinity, Decentraland
Real Estate Contracts	For property tokenization, ownership transfer	Tokenized property platforms
NFT Contracts	Non-Fungible Tokens: digital art, collectibles	ERC721, ERC1155
Healthcare Contracts	Secure patient data, insurance processing	HealthChain, MedRec

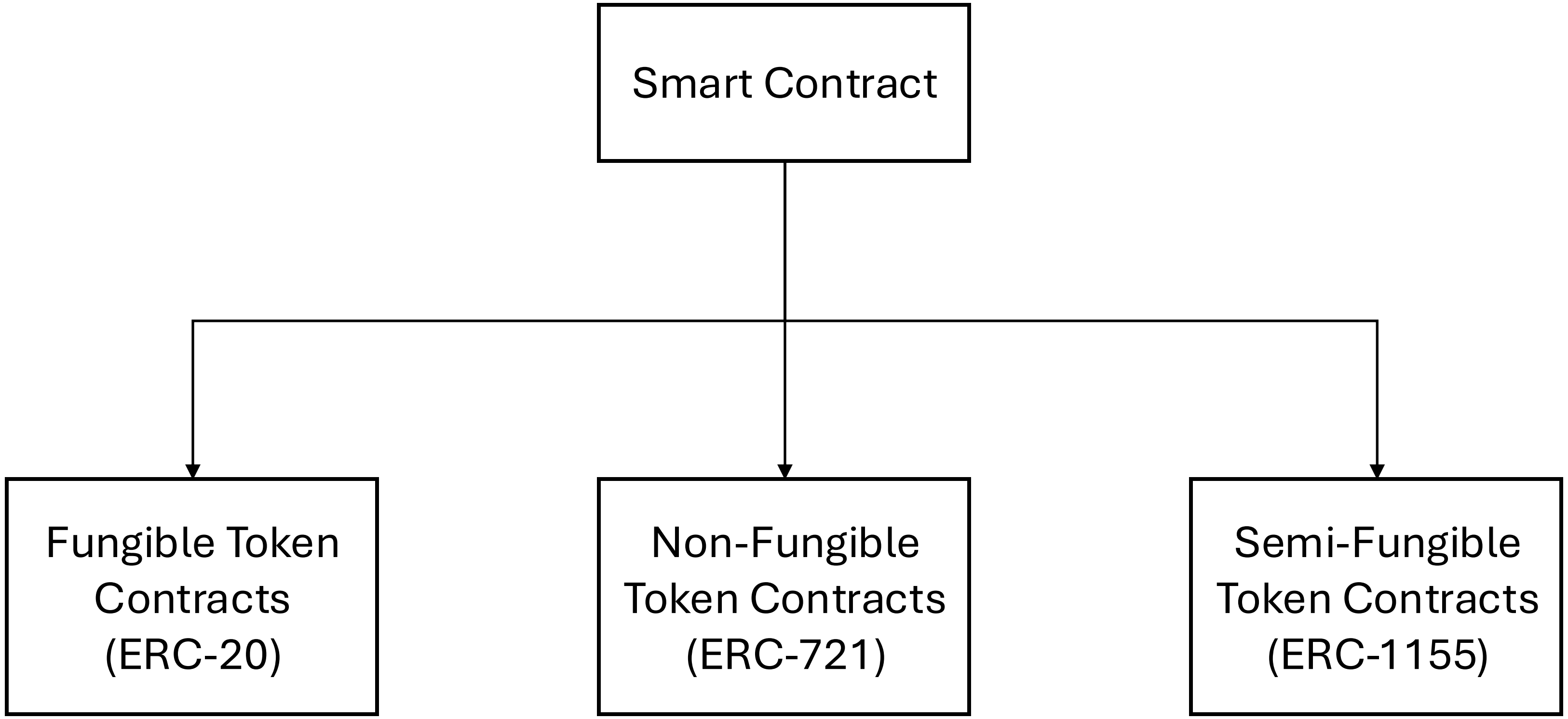
Based on Platform/Execution Environment

Type	Platform Example	Feature Highlight
Ethereum Smart Contracts	Solidity-based	Gas-based execution, high adoption
Hyperledger Chaincode	Go/JavaScript-based	Permissioned, enterprise-grade
Solana Smart Contracts	Rust-based	High-speed, low-latency
Cardano Plutus Contracts	Haskell-based	Formal verification

Real-World Use Cases of Smart Contract Features

Domain	Use Case	Feature Used
Finance	Automatic loan repayment	Self-execution, conditions
Insurance	Payout upon disaster verification	Oracle + smart contract
Supply Chain	Traceability and delivery confirmation	Event-logging + conditions
Real Estate	Tokenized property sales	Composability + immutability

Smart Contract Types



Fungible Token Contracts (ERC-20)

- **Definition:** Tokens that are interchangeable and identical in value.
- **Examples:** ETH, IMX, USDT
- **Use Cases:**
  - Staking
  - Voting in DAOs
  - Trading and liquidity mining
  - Game rewards and user incentives

Non-Fungible Token Contracts (ERC-721)

- **Definition:** Tokens that are unique and non-interchangeable.
- **Examples:** Bored Apes, Gods Unchained cards
- **Use Cases:**
  - Ownership & authenticity of digital/physical items
  - Gaming assets (e.g., skins, weapons)
  - Creator royalties
  - Event tickets & access passes
  - Crowdfunding and loyalty programs

Semi-Fungible Tokens (ERC-1155)

- **Definition:** Hybrid standard for both fungible and non-fungible assets.
- **Use Cases:** Popular in gaming (e.g., in-game currencies and unique weapons)

Key Components of Token Smart Contracts

Component	Fungible Tokens (ERC-20)	Non-Fungible Tokens (ERC-721)
Supply Info	totalSupply() - total tokens created	totalSupply() - total tokens minted (ERC721Enumerable)
Ownership Mapping	balanceOf(address) - tokens owned by user	ownerOf(tokenID) - who owns a specific token
Minting	mint(address, amount)	mint(tokenID, address)
Metadata	Not always required	tokenURI(tokenID) - points to metadata JSON
Transfer	transfer(address, amount)	transferFrom(address, address, tokenID)
Approval	approve(address, amount)	approve(address, tokenID)

Example: NFT Contract (ERC-721)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "./Mintable.sol";

contract Asset is ERC721, Mintable {
    constructor(
        address _owner,
        string memory _name,
        string memory _symbol,
        address _imx
    ) ERC721(_name, _symbol) Mintable(_owner, _imx) {}

    function _mintFor(
        address user,
        uint256 id,
        bytes memory
    ) internal override {
        _safeMint(user, id);
    }
}
```

Mintable.sol (Supporting Contract)

- Handles authorization (only IMX can mint)
- Emits event: AssetMinted(address to, uint256 id, bytes blueprint)
- Stores blueprint metadata in a mapping for each token ID

L1 vs L2 Minting on Immutable X

- **FTs:** Must be minted on Layer 1 (Ethereum), then deposited to L2
- **NFTs:** Can be minted directly on Layer 2 using contracts like Asset.sol

Real-world Use Cases

Fungible Tokens (FTs)

- **Staking:** Lock tokens to earn rewards
- **DAO Voting:** Governance via token-weighted votes
- **Rewards:** Incentivize participation in games, DeFi, or communities

Non-Fungible Tokens (NFTs)

- **Ownership:** Prove authenticity of digital assets
- **Gaming:** Transferable and tradable in-game items
- **Royalties:** Earn recurring income from secondary sales
- **Access Control:** Use NFTs as event tickets or premium passes

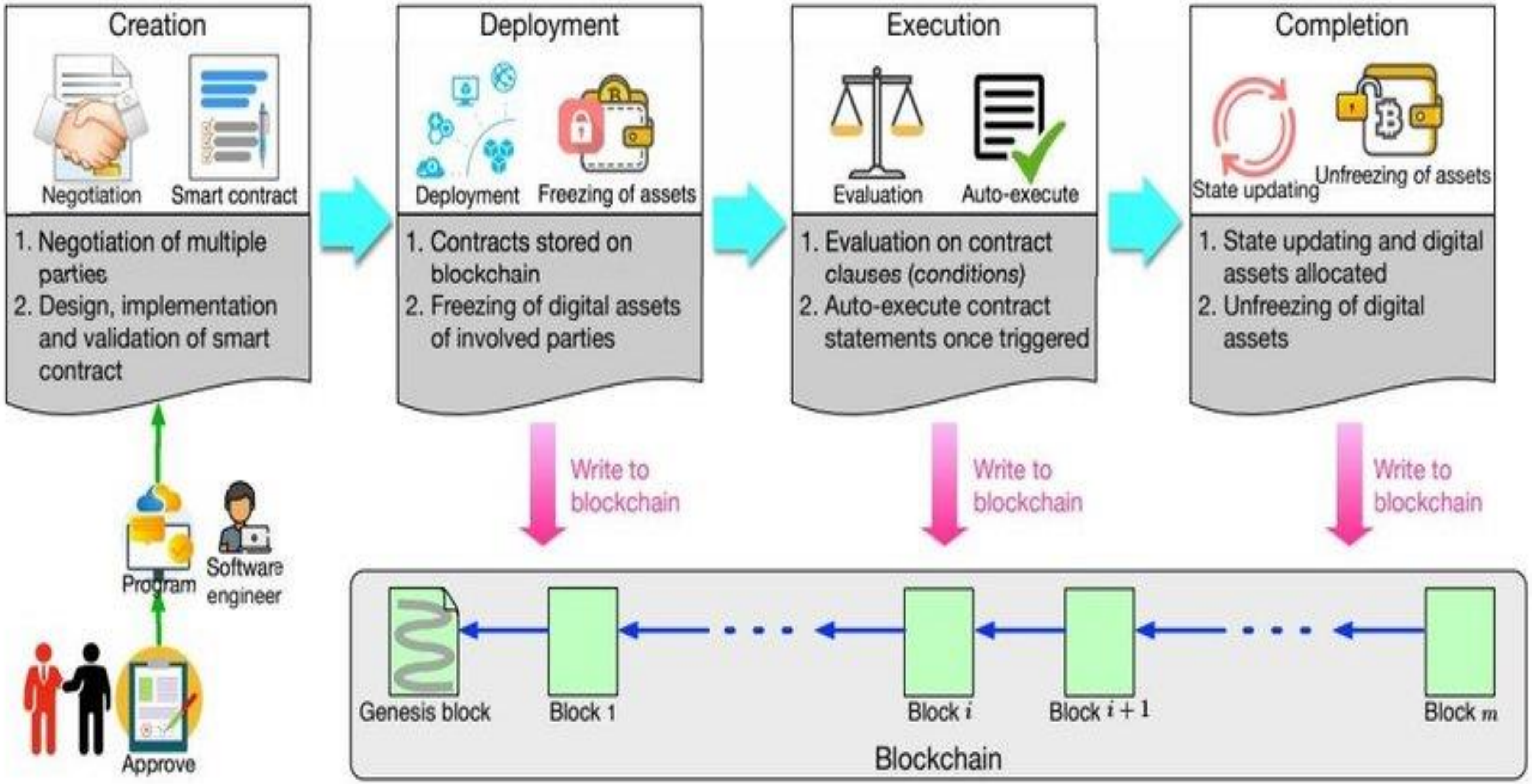
Lifecycle of a Smart Contract

- The **lifecycle of a smart contract** refers to the complete journey of a smart contract from creation to eventual deactivation (or indefinite operation).
- This process is particularly relevant for platforms like Ethereum and includes both developer and user interactions.

Lifecycle of a Smart Contract

1. Design & Development
2. Compilation
3. Deployment
4. Initialization
5. Execution & Interaction
6. Upgrading (If Applicable)
7. Termination (Optional)

Lifecycle of a Smart Contract



Lifecycle of a Smart Contract

1. Design & Development

- **What happens?**
  - The smart contract logic is written using a programming language (e.g., Solidity for Ethereum).
  - Developers define functions, permissions, state variables, and interactions with other contracts.
- **Tools used:**
  - IDEs like Remix, Hardhat, Truffle
  - Testing frameworks: Mocha, Chai

Lifecycle of a Smart Contract

2. Compilation

- **What happens?**
  - The source code is compiled into bytecode (machine-readable format) and an **ABI (Application Binary Interface)**.
- **Output:**
  - Bytecode (for EVM deployment)
  - ABI (used to interact with the contract through wallets or web apps)

Lifecycle of a Smart Contract

3. Deployment

- **What happens?**
  - The compiled contract is deployed to a blockchain.
  - This is done by submitting a **deployment transaction** which includes the bytecode and constructor arguments.
- **Gas Fee Required?** Yes
- **Result:**
  - A **unique address** is assigned to the smart contract on the blockchain.

Lifecycle of a Smart Contract

4. Initialization (Optional)

- **What happens?**
  - Some contracts include initialize() functions for setting up roles, owners, or dependencies (used especially in upgradable contracts).
- **State is set.**

Lifecycle of a Smart Contract

5. Execution & Interaction

- **What happens?**
  - Users or other contracts **call the contract's functions** (e.g., mint(), transfer(), vote()).
  - This can change the contract's internal state or emit events.
- **Types of function calls:**
  - **Read-only/view** (no gas, off-chain)
  - **State-changing** (requires gas, updates blockchain)

Lifecycle of a Smart Contract

6. Upgrading (If Applicable)

- **What happens?**
  - Some contracts follow **proxy patterns** (e.g., OpenZeppelin's upgradable contracts) where logic can be changed without altering the contract's address.
- **Reason:** Fix bugs, add features, or improve efficiency.
- 🚫 Regular contracts are immutable unless a proxy pattern is used!

Lifecycle of a Smart Contract

7. Termination (Optional)

- **What happens?**
  - The contract can be destroyed using the selfdestruct() function.
  - This **removes the code and storage** from the blockchain.
- **Why terminate?**
  - Contract is obsolete, buggy, or a time-limited service.
- ⚠️ Once destroyed, it **cannot be recovered**.

Smart Contract Lifecycle Example: DeFi Staking Contract

Goal: **Allow users to stake tokens and earn rewards over time.**

1. Design & Development

- **Define functions:**
  - stake(uint amount)
  - withdraw(uint amount)
  - claimRewards()
  - calculateReward(address user)
- **Data structures:**
  - Mapping of user → staked balance
  - Timestamps for reward calculation
- **Language/Platform:**
  - Solidity on Ethereum or Polygon

2. Compilation

- The smart contract is compiled using tools like:
  - **Remix IDE**, **Hardhat**, or **Truffle**
- Outputs:
  - **Bytecode**: For EVM
  - **ABI**: For frontend/web3 interaction

Smart Contract Lifecycle Example: DeFi Staking Contract

```
/// @notice Stake `amount` of stakeToken into the pool.
function stake(uint256 amount) external nonReentrant {
    require(amount > 0, "ZERO_AMOUNT");
    _updateUser(msg.sender);

    totalStaked += amount;
    balanceOf[msg.sender] += amount;

    stakeToken.safeTransferFrom(msg.sender, address(this), amount);
    emit Staked(msg.sender, amount);
}
```

Smart Contract Lifecycle Example: DeFi Staking Contract

```
/// @notice Claim all accrued rewards (rewardToken).
function claimRewards() external nonReentrant {
    _updateUser(msg.sender);
    uint256 reward = rewards[msg.sender];
    require(reward > 0, "NO_REWARDS");
    rewards[msg.sender] = 0;

    rewardToken.safeTransfer(msg.sender, reward);
    emit RewardPaid(msg.sender, reward);
}
```

Smart Contract Lifecycle Example: DeFi Staking Contract

```
/// @notice Claim all accrued rewards (rewardToken).
function claimRewards() external nonReentrant {
    _updateUser(msg.sender);
    uint256 reward = rewards[msg.sender];
    require(reward > 0, "NO_REWARDS");
    rewards[msg.sender] = 0;

    rewardToken.safeTransfer(msg.sender, reward);
    emit RewardPaid(msg.sender, reward);
}
```

Smart Contract Lifecycle Example: DeFi Staking Contract

3. Deployment

- Deployed to Ethereum using Metamask or scripts.
- Deployer provides:
  - Reward rate
  - Token address (ERC-20)
- Once deployed, a unique **contract address** is created.
- The staking contract is now live on-chain.

4. Initialization

- The contract may initialize:
  - Accepted token (e.g., DAI)
  - Reward rate (e.g., 5% annually)
  - Admin roles (e.g., contract owner)

Smart Contract Lifecycle Example: DeFi Staking Contract

5. Execution & Interaction

- **Users** call:
  - stake(amount) → locks tokens
  - claimRewards() → retrieves earned tokens
  - withdraw(amount) → unstakes
- Each function call is a **transaction**, consuming **gas**.
- Backend/frontend uses web3.js or ethers.js to interact.

6. Upgrade (Optional)

- If the contract uses **proxy pattern**, developers can:
  - Change reward formula
  - Add penalty for early withdrawal
- Done via admin-controlled logic upgrade
- Without a proxy, the contract is immutable.

Smart Contract Lifecycle Example: DeFi Staking Contract

7. Termination (Optional)

- The admin may call selfdestruct() if:
  - Project is discontinued
  - Rewards exhausted
- Transfers any remaining funds to admin or DAO treasury
- After termination, no functions can be called, and users can no longer stake or withdraw.

Smart Contract Lifecycle Example: DeFi Staking Contract

Stage	Action Example
Design	Define staking, withdrawing, rewards logic
Compile	Compile with Hardhat → Bytecode & ABI
Deploy	Deploy with MetaMask → Get contract address
Interact	Users stake tokens & earn rewards
Upgrade	Add early withdrawal fee (via proxy)
Terminate	Shutdown with selfdestruct()

References

- https://ethereum.org/en/developers/docs/smart-contracts/
- https://docs.immutable.com/x/anatomy-smart-contract/