Topic 4.1: Smart Contracts

Code as Agreement

Joerg Osterrieder

Digital Finance

2025

Learning Objectives

By the end of this topic, you will be able to:

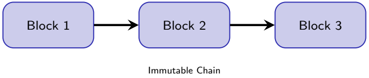
1. **Define** what a smart contract is and how it differs from traditional contracts
2. **Explain** how smart contracts execute on blockchain networks
3. **Understand** the Ethereum Virtual Machine (EVM—a shared computer that runs programs) and gas mechanism (a fee to prevent spam)
4. **Read** basic Solidity smart contract code with simple syntax guidance
5. **Identify** key limitations: oracle problem, immutability, reentrancy (when a contract is called back before it finishes)
6. **Analyze** what "trustless" really means in practice

Hands-on: NB08 – Smart Contract Interaction

Prerequisites: Blockchain Foundations Recap

From Day 3 – Key Concepts:

- **Blockchain**: Distributed, immutable ledger (shared record book)
- **Consensus**: Agreement without central authority (voting without a boss)
- **Cryptography**: Hashes secure data integrity (digital fingerprints)
- **Wallets**: Private/public key pairs for identity (password + username)
- **Transactions**: Signed messages changing state (sending a signed check)



Why this matters:

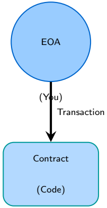
- Smart contracts *live on* blockchains
- They inherit blockchain's security properties
- Transactions trigger contract execution

Ethereum vs. Bitcoin:

- Bitcoin: "Programmable money" (limited scripts)
- Ethereum: "World computer"—can run any program you can imagine, like a universal calculator vs a basic one

Two Types of Accounts:

1. **Personal Wallets (Externally Owned Accounts - EOA)**
   - Controlled by private keys (humans/wallets)
   - Can initiate transactions
2. **Smart Program Wallets (Contract Accounts)**
   - Controlled by code (smart contracts)
   - Can only respond to transactions



Key Point

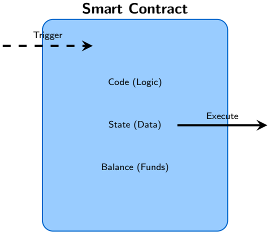
Contracts cannot act on their own—they only execute when triggered by transactions.

Definition

A **smart contract** is a program stored on a blockchain that automatically executes when predetermined conditions are met.

Key Properties:

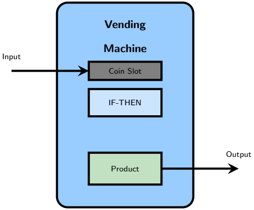
- **Deterministic**: Same input always produces same output (like a calculator)
- **Immutable**: Once deployed, code cannot be changed (carved in stone)
- **Transparent**: Anyone can verify the code (like a glass box)
- **Self-executing**: No intermediary needed



Original Concept

Nick Szabo coined "smart contracts" in 1994—long before Bitcoin!

"A computerized transaction protocol that executes the terms of a contract."



The Vending Machine Analogy:

1. Insert coins (input)
2. Select product (function call)
3. Receive product (output)
4. No trust in seller needed!

The machine *enforces the rules* automatically.

Key Insight

Rules are embedded in the mechanism itself.

Traditional Contract vs. Smart Contract

Aspect	**Traditional Contract**	**Smart Contract**
Enforcement	Courts, lawyers	Code execution
Trust	Counterparties, institutions	Cryptographic verification
Execution	Manual, subject to delay	Automatic, instant
Amendment	Negotiation, paperwork	Requires new deployment
Cost	High (intermediaries)	Low (gas fees only)
Transparency	Private documents	Public, auditable code

Important Distinction

"Trustless" means you don't need to trust *counterparties*—but you still need to trust the *code*, the *blockchain*, and the *oracles*.

The "Code is Law" Principle

Definition

The code's logic is the absolute and final arbiter—whatever the code does is what happens, regardless of intent.

Implications:

- No appeals court for bugs
- No "that's not what I meant"
- Code executes *exactly* as written
- Ambiguity is impossible (deterministic)

The DAO Hack (2016)

What is reentrancy?

- Like an ATM giving you money twice because it didn't update your balance fast enough

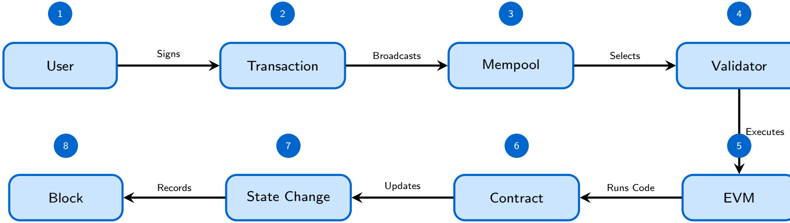
What happened:

- Attacker exploited reentrancy bug
- Drained $60 million in ETH
- Code executed exactly as written
- "Legal" according to the code

Result: Ethereum hard-forked to reverse the hack—creating Ethereum Classic.

With great automation comes great responsibility.

Smart Contract Execution Flow



In Simple Terms

(1) You send a request **(2)** Network checks it **(3)** Contract executes

1. User creates and signs transaction calling contract function
2. Transaction broadcast to network
3. Transaction waits in mempool[a]
4. Validator[b] selects transaction for block
5. Ethereum Virtual Machine (EVM) executes bytecode[c]
6. Contract logic runs with provided inputs
7. State changes recorded (balances, storage)
8. Changes finalized in blockchain block

[a] **Mempool**: a waiting room for unconfirmed transactions. [b] **Validator**: a network participant that confirms transactions. [c] **Bytecode**: machine-readable instructions.

The Ethereum Virtual Machine (EVM)

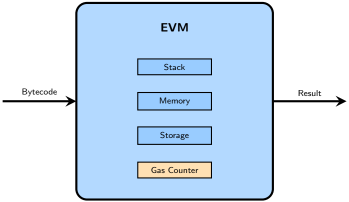
Key Idea

You don't need to understand the technical details. The key idea: the EVM is a **virtual computer that every Ethereum node runs identically**, so everyone agrees on the result.

What is the EVM?

A runtime environment that executes smart contract bytecode in a **deterministic**, **isolated** environment across all network nodes.

Think of it as a *shared computer that runs programs* the same way everywhere.



Key Properties:

- **Sealed box**: Can only do math and save results (like a sealed box that cannot access your computer)
- **Deterministic**: Same output everywhere
- **Metered**: Every operation costs gas

Why "Virtual"?

- Not a physical machine
- Runs identically on all nodes
- Enables consensus on state

Gas: The Fuel of Smart Contracts

What is Gas?

A unit measuring computational effort required to execute operations on the EVM.

Think of gas as a *fee to prevent spam*, like paying postage for a letter—heavier letters cost more.

Why Gas Exists:

- **Prevent spam**: Makes attacks expensive
- **Halt infinite loops**: Transactions run out of gas
- **Incentivize efficiency**: Cheaper code = lower fees
- **Pay validators**: Compensation for computation

Gas Cost Examples:

Operation	**Gas**
Addition	3
Multiplication	5
SHA3 Hash	30
Balance check	700
Storage write	**20,000**
Contract creation	32,000+

Key Insight

Storage is *extremely* expensive—10,000x more than arithmetic!

Gas Price vs. Gas Limit

Gas Price (Gwei):

- How much you pay *per unit* of gas
- Set by the user
- Higher price = faster inclusion
- 1 Gwei $= 10^{-9}$ ETH

Gas Limit:

- Maximum gas units allowed
- Set by the user
- If exceeded, transaction reverts
- Unused gas is refunded

Transaction Fee Formula

$$\text{Fee} = \text{Gas Used} \times \text{Gas Price}$$

Example:

- Gas Used: 21,000 (simple transfer)
- Gas Price: 20 Gwei
- Fee: $21{,}000 \times 20 = 420{,}000$ Gwei
- $= 0.00042$ ETH ($\approx$ \$1.50)

If Out of Gas

Transaction reverts but gas is NOT refunded—you still pay for failed computation.

Solidity: The Smart Contract Language

What is Solidity?

The main programming language used to write smart contracts on Ethereum. It was designed specifically for blockchain use.

No Coding Required

You won't need to write Solidity in this course. The following slides show code examples so you can *read* what a smart contract looks like.

Key Features:

- Syntax similar to JavaScript/C++
- Compiles to EVM bytecode (machine-readable instructions)
- Built-in support for ETH transfers

Other Smart Contract Languages:

Language	Platform
Solidity	Ethereum, BSC
Vyper	Ethereum (Python-like)
Rust	Solana, Near
Move	Sui, Aptos
Michelson	Tezos

Focus

Solidity is the most widely used—understanding it transfers to other platforms.

Anatomy of a Smart Contract (Solidity)

Optional: Code Deep Dive

The following slides show real smart contract code. This is **optional material** for those interested in programming. The key concepts are explained in plain language alongside each code example.

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

contract SimpleEscrow {
    address public buyer;
    address public seller;
    uint256 public amount;
    bool public released;

    constructor(address _seller) payable {
        buyer = msg.sender;
        seller = _seller;
        amount = msg.value;
    }

    function release() external {
        require(msg.sender == buyer, "Only buyer can release");
        require(!released, "Already released");
        released = true;
        payable(seller).transfer(amount);
    }
}
```

Plain-English Summary

This is a simple **escrow** (a middleman that holds money). A buyer deposits funds; only the buyer can release them to the seller. The contract holds the money until the buyer says "release." No bank or lawyer needed.

Understanding the Contract Structure

If the code on the previous slide looked intimidating, don't worry—here is a plain-English guide to reading it.

Reading Code 101

address = wallet ID
uint256 = positive number
bool = true/false
require() = "check that this is true"
msg.sender = "who called this function"

1. **License & Version:**
   - `SPDX-License-Identifier`: Legal license
   - `pragma solidity`: Compiler version
2. **State Variables:**
   - `address`: Ethereum addresses
   - `uint256`: Unsigned integers
   - `bool`: True/false values
   - Stored permanently on blockchain
3. **Constructor:**
   - Runs once at deployment
4. **Functions:**
   - `external`: Called from outside only
   - `public`: Called from anywhere
   - `view`: Reads state, no changes
   - `pure`: No state access at all
5. **Access Control:**
   - `require()`: Validates conditions
   - `msg.sender`: Who called the function
   - `msg.value`: ETH sent with call

Key Pattern

Checks-Effects-Interactions: Validate first, update state, then interact with external contracts.

State Variables vs. Local Variables

Optional Advanced Material

State variables are **permanent data stored on the blockchain** (like entries in a shared database). Local variables exist **only during one function call** and are then discarded (like scratch paper).

State Variables:

- Stored **permanently** on blockchain
- Persist between function calls
- **Expensive**: 20,000 gas to write
- Declared at contract level

```
contract Example {
    // State variable
    uint256 public balance;
}
```

Local Variables:

- Exist only during function execution
- Stored in memory (cheap)
- Discarded after function ends
- Declared inside functions

```
function calculate() public {
    // Local variable
    uint256 temp = 100;
}
```

Gas Optimization Tip

Minimize state variable writes. Use local variables for intermediate calculations, then write the final result once.

Function Visibility and Modifiers

Optional Advanced Material

Smart contracts can have **public functions** (anyone can call them) and **private functions** (only the contract itself can call them)—like the difference between a shop's front door and its staff-only entrance.

Visibility Modifiers:

Modifier	Access
`public`	Anyone (internal + external)
`external`	Only from outside the contract
`internal`	This contract + inheriting contracts
`private`	Only this contract

State Modifiers:

Modifier	Meaning
`view`	Reads state, doesn't modify
`pure`	No state read or write
`payable`	Can receive ETH
(none)	Can modify state

Cost Note

`view` and `pure` functions are **free** when called externally (off-chain). They only cost gas when called by a transaction (on-chain).

Custom Modifiers: Reusable Access Control

Optional Advanced Material

Modifiers are **reusable checks** that run before a function executes—like a bouncer checking IDs before letting someone into a club. If the check fails, the function never runs.

```
contract Owned {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Not the owner");
        _; // Function body executes here
    }

    function withdraw() external onlyOwner {
        // Only owner can call this
        payable(owner).transfer(address(this).balance);
    }
}
```

Modifier Pattern

- Modifiers run **before** the function body (at the underscore)
- Commonly used for: access control, input validation, reentrancy guards
- Creates cleaner, more maintainable code

Events: Logging to the Blockchain

Optional Advanced Material

Events are **notifications** that a smart contract sends out when something happens—like a receipt you get after a purchase. External applications can "listen" for these events.

```
contract Token {
    event Transfer(
        address indexed from,
        address indexed to,
        uint256 value
    );

    function transfer(address to, uint256 amount) external {
        // ... transfer logic ...
        emit Transfer(msg.sender, to, amount);
    }
}
```

Why Events?

- **Cheap**: 375 gas base (vs 20,000 for storage)
- **Indexed**: Efficient off-chain searching
- **Notifications**: dApps can listen for events

Important Limitation:

- Events are **not** accessible from contracts
- They're for external observers only
- Cannot trigger on-chain actions

Deployment Details:

1. **Compile**: Solidity $\rightarrow$ EVM bytecode + ABI[a]
2. **Create Transaction**: Send bytecode to address 0x0 (null)
3. **Execute Constructor**: Runs once, sets initial state
4. **Generate Address**: From deployer address + nonce[b] (deterministic)
5. **Store Code**: Runtime bytecode stored at new address—**immutable**

[a] **ABI** = Application Binary Interface—a menu listing what the contract can do (its public functions and their inputs/outputs).

[b] **Nonce** = a counter ensuring each transaction is unique, preventing replay attacks.

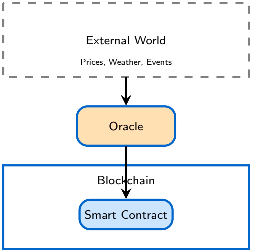
The Oracle Problem

The Challenge

Smart contracts cannot access external data—they only know what's on the blockchain.

Examples requiring oracles:

- Price feeds (ETH/USD)
- Weather data for insurance
- Sports scores for betting
- Real-world asset values



Oracle Solutions:

- Chainlink (decentralized)
- API3, Band Protocol
- Optimistic oracles (UMA)

Oracle Trust Tradeoffs

Centralized Oracle:

- Single data source
- Fast and cheap
- Single point of failure
- Trust one entity

Decentralized Oracle (Chainlink):

- Multiple independent nodes
- Aggregated data (median)
- No single point of failure
- More expensive

Oracle Attack Vectors

- **Manipulation**: Feed false data
- **Flash loan attacks**: Manipulate on-chain prices
- **Latency**: Stale data exploitation

Key Insight

Oracles are the bridge between on-chain and off-chain—and bridges are often the weakest link.

Smart Contract Risks and Vulnerabilities

These risks explain why smart contract security is so important. You don't need to understand the technical details of each attack—focus on the big picture.

Technical Risks:

- **Bugs**: Code is immutable—bugs are forever
- **Reentrancy**: The DAO hack ($60M, 2016)
- **Integer overflow**: Pre-0.8 Solidity
- **Oracle manipulation**: Flash loan attacks
- **Front-running**: MEV extraction

Gas Considerations:

- Every operation costs gas
- Complex logic = expensive
- Storage is most expensive
- Out-of-gas reverts transaction

Design Limitations:

- Cannot handle ambiguity
- No subjective judgments
- Cannot initiate actions
- Limited to on-chain data

The Immutability Paradox

Immutability provides security guarantees but makes bug fixes impossible.

Solutions: Proxy patterns, upgradeable contracts—but these reintroduce trust assumptions.

Optional: Technical Deep Dive

The reentrancy attack exploits a **timing flaw**: a malicious contract calls back into the victim before the first transaction finishes, draining funds repeatedly. The fix: **always update your records BEFORE sending money.**

Vulnerable Code:

```
function withdraw() external {
  uint256 bal = balances[msg.sender];
  // DANGER: External call first
  msg.sender.call{value: bal}("");
  // State update AFTER call
  balances[msg.sender] = 0;
}
```

Safe Code (Checks-Effects-Interactions):

```
function withdraw() external {
  uint256 bal = balances[msg.sender];
  // Effect: Update state FIRST
  balances[msg.sender] = 0;
  // Interaction: External call LAST
  msg.sender.call{value: bal}("");
}
```

The Attack:

1. Attacker calls `withdraw()`
2. Contract sends ETH to attacker
3. Attacker's fallback re-calls `withdraw()`
4. Balance not yet updated—sends again!
5. Loop until drained

Prevention

- Update state before external calls
- Use ReentrancyGuard modifier
- Prefer `transfer()` (limited gas)

Hands-On Exercise: NB08 – Smart Contract Interaction

Notebook Objectives

1. **Connect** to Ethereum testnet using web3.py
2. **Read** contract state (balances, variables)
3. **Call** contract functions
4. **Observe** state changes and events
5. **Understand** transaction receipts and gas usage

What You'll Need:

- Python with web3.py installed
- Testnet ETH (from faucet)
- Deployed contract address
- Contract ABI (interface)

Learning Outcomes:

- Read contract data without gas
- Sign and send transactions
- Interpret transaction results
- Debug failed transactions

Optional: For Students with Programming Experience

In the notebook, all code runs automatically. You only need to **observe the outputs and answer the analysis questions**—no programming required.

Reading Contract State (Free):

```
from web3 import Web3

# Connect to network
w3 = Web3(Web3.HTTPProvider(url))

# Load contract
contract = w3.eth.contract(
    address=address,
    abi=abi
)

# Read state (no gas)
balance = contract.functions
    .balanceOf(account).call()
```

Writing to Contract (Costs Gas):

```
# Build transaction
tx = contract.functions.transfer(
    to_address, amount
).build_transaction({
    'from': my_address,
    'nonce': w3.eth.get_nonce(
        my_address
    ),
    'gas': 100000,
    'gasPrice': w3.eth.gas_price
})

# Sign and send
signed = w3.eth.account
    .sign_transaction(tx, key)
tx_hash = w3.eth.send_raw
    _transaction(signed.rawTx)
```

Complete exercise in NB08 Jupyter notebook

Discussion: Real-World Smart Contract Applications

Already in Production:

- **DeFi**: Lending, trading, yield
- **NFTs**: Digital ownership
- **DAOs**: Governance voting
- **Stablecoins**: Algorithmic pegging
- **Insurance**: Parametric payouts

Emerging Applications:

- Supply chain tracking
- Real estate tokenization
- Identity verification
- Royalty distribution

Discussion Questions:

1. What types of agreements are *best suited* for smart contracts?
2. What types of agreements should *never* be smart contracts?
3. How do you balance immutability with the need for bug fixes?

Think About

What makes a contract "smart" vs. just automated?

When Smart Contracts Are NOT the Answer

Poor Fit:

- **Subjective decisions**: "Was the work satisfactory?"
- **Ambiguous terms**: Contracts requiring interpretation
- **Changing conditions**: Agreements that need flexibility
- **Private data**: Blockchain is public
- **Low-value transactions**: Gas fees exceed value

Good Fit:

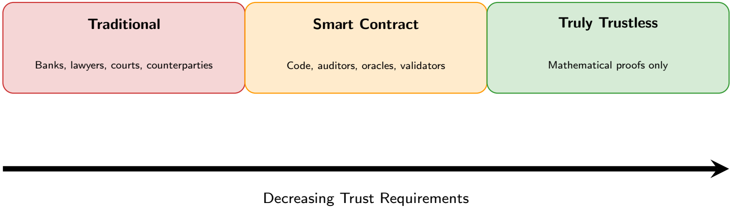
- **Binary outcomes**: Yes/no, on-time/late
- **Measurable conditions**: Price thresholds, dates
- **High trust cost**: When intermediaries are expensive
- **Transparency needed**: Public verification matters
- **Composability**: Building on other contracts

Rule of Thumb

If the contract requires human judgment or interpretation, a smart contract alone is insufficient. Consider hybrid approaches: smart contracts for execution + arbitration mechanisms for disputes.

What “Trustless” Really Means

Trust Spectrum



Key Insight

Smart contracts shift trust from *institutions* to *code and cryptography*. This is valuable—but it’s a different kind of trust, not the absence of trust.

Smart Contract Security Best Practices

Before Deployment:

1. **Thorough testing**: Unit + integration tests
2. **Formal verification**: Mathematical proofs
3. **Multiple audits**: Independent security reviews
4. **Bug bounties**: Incentivize white-hat hackers
5. **Testnet deployment**: Extended testing period

Design Patterns:

- Checks-Effects-Interactions
- Reentrancy guards
- Access control (OpenZeppelin)
- Timelock for critical changes

After Deployment:

- **Monitoring**: Watch for anomalies
- **Pause mechanism**: Emergency stop
- **Upgrade path**: Proxy patterns if needed
- **Insurance**: Cover potential losses

Audit Reality Check

"Audited" does not mean "bug-free."

Many hacked protocols had multiple audits. Audits reduce risk but don't eliminate it.

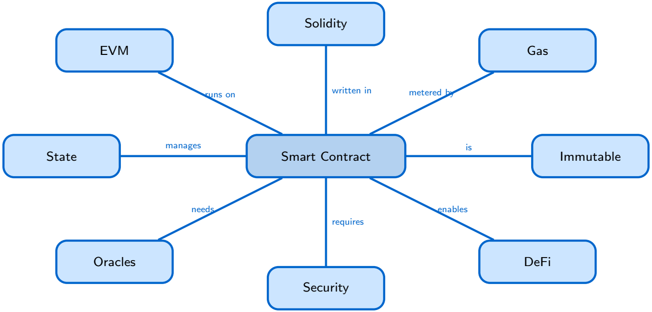
Executive Summary

Smart Contracts: Code as Agreement

- **Definition**: Self-executing programs on blockchain that automatically enforce contract terms
- **Key Properties**: Deterministic, immutable, transparent, self-executing
- **Execution**: User transaction $\rightarrow$ EVM executes bytecode $\rightarrow$ State changes recorded
- **Gas**: Computational fuel that prevents spam and incentivizes efficiency
- **Oracle Problem**: Contracts cannot access external data directly
- **Risks**: Immutability (bugs are forever), reentrancy, oracle manipulation
- **"Trustless"**: Shifts trust from institutions to code—different trust, not no trust

Smart contracts enable programmable, automatic agreements—but require extreme care in development.

Concept Map: Smart Contracts



Key Terms (1/2)

Smart Contract

- Self-executing code on blockchain
- Automatically enforces agreement terms

EVM (Ethereum Virtual Machine)

- Runtime environment for bytecode
- Deterministic execution across nodes

Solidity

- High-level programming language
- Compiles to EVM bytecode

Gas

- Unit of computational effort
- Prevents spam, funds validators

State Variables

- Permanently stored on blockchain
- Expensive to write (20,000 gas)

msg.sender / msg.value

- Caller's address / ETH sent
- Essential for access control

Modifier

- Reusable function guard
- Common: `onlyOwner`, `nonReentrant`

Events

- Cheap logging mechanism
- For off-chain monitoring

Key Terms (2/2)

Oracle

- Bridge between off-chain and on-chain
- Provides external data to contracts

Immutability

- Code cannot be changed post-deployment
- Bugs are permanent

Reentrancy

- Vulnerability: recursive calls drain funds
- Prevention: Checks-Effects-Interactions

Proxy Pattern

- Enables contract upgradability
- Separates storage from logic

ABI (Application Binary Interface)

- Contract's public interface definition
- Required to interact with contract

Constructor

- Runs once at deployment
- Sets initial state

payable

- Function can receive ETH
- Required for deposits

view / pure

- `view`: reads state only
- `pure`: no state access

Common Misconceptions

Myth 1: "Trustless" = No Trust

- Wrong: You trust the code, oracles, auditors, and blockchain
- Reality: Trust is shifted, not eliminated

Myth 2: "Audited" = Safe

- Wrong: Many hacked protocols had multiple audits
- Reality: Audits reduce risk, don't eliminate it

Myth 3: Smart Contracts Can Do Anything

- Wrong: They can't access external data or initiate actions
- Reality: They're limited to on-chain data and reactive execution

Myth 4: Code is Law (Absolute)

- Wrong: Social consensus can override (Ethereum/ETC fork)
- Reality: Code is law until humans decide otherwise

Myth 5: Smart Contracts Are Legally Binding

- Wrong: Legal status varies by jurisdiction
- Reality: Code execution $\neq$ legal enforceability

Myth 6: Immutable = Secure

- Wrong: Immutability locks in bugs too
- Reality: It's a double-edged sword

Self-Assessment Questions (1/2)

Question 1: Smart Contract Definition

What is the fundamental definition of a smart contract?

A. A legal agreement written in computer code that can be read by lawyers
B. Self-executing code stored on a blockchain that automatically enforces contract terms without intermediaries
C. A digital document that requires manual verification by network validators
D. An AI-powered system that negotiates contract terms between parties

Self-Assessment Questions (1/2)

Question 1: Smart Contract Definition

What is the fundamental definition of a smart contract?

A. A legal agreement written in computer code that can be read by lawyers
B. Self-executing code stored on a blockchain that automatically enforces contract terms without intermediaries
C. A digital document that requires manual verification by network validators
D. An AI-powered system that negotiates contract terms between parties

Answer: B
A smart contract is self-executing code stored on a blockchain that automatically enforces the terms of a contract without requiring trusted intermediaries. Like Szabo's vending machine—it enforces rules automatically.

Self-Assessment Questions (2/2)

Question 2: What is Gas?

What is "gas" in the context of Ethereum smart contracts?

A. A physical resource consumed by mining hardware
B. A unit measuring computational effort required to execute operations on the EVM
C. The fuel token used to power Ethereum's consensus mechanism
D. A type of cryptocurrency alternative to Ether

Answer: B — Gas measures computational effort. Storage writes cost 20,000 gas; addition costs 3 gas.

Question 3: Modifiers in Solidity

What is the purpose of modifiers in Solidity?

A. To modify the return value of functions automatically
B. To create reusable access control and validation logic that runs before function execution
C. To adjust gas prices based on network congestion
D. To convert function outputs to different data types

Answer: B — Modifiers like `onlyOwner` provide reusable guards before function execution.

What's Next: Topic 4.2 – DeFi Primitives

Coming Up:

- **Lending Protocols**: Aave, Compound
- **Decentralized Exchanges**: AMMs, Uniswap
- **Yield Farming**: Liquidity mining
- **Composability**: "Money Legos"

How Topics Connect:
Smart contracts (T4.1) are the *building blocks*. DeFi (T4.2) shows what you can *build with them*.

How can smart contracts replicate what banks do—but without the bank?

b0.48 **Official Documentation:**

- Solidity Docs: `docs.soliditylang.org`
- Ethereum.org: `ethereum.org/developers`
- OpenZeppelin: `docs.openzeppelin.com`

Learning Platforms:

- CryptoZombies (interactive tutorials)
- Ethernaut (security challenges)
- Speedrun Ethereum (build projects)

Development Tools:

- Remix IDE (browser-based)
- Hardhat (Node.js framework)
- Foundry (Rust-based testing)

Security Resources:

- SWC Registry (vulnerabilities)
- Consensys Security Best Practices
- Trail of Bits publications

Academic Papers:

- Szabo, N. (1994) "Smart Contracts"
- Buterin, V. (2014) "Ethereum Whitepaper"
- Chen et al. (2020) "Survey of Smart Contract Security"

Course Notebook:

- **NB08**: Smart Contract Interaction

b

Questions?

Key Takeaway:

Smart contracts are powerful automation tools that shift trust from institutions to code—but they require extreme care *in development because bugs are forever.*

Next Topic: T4.2 – DeFi Primitives: Lending, Trading, and Yield