Ethereum & Smart Contracts

Lesson 3 of 4: Cryptocurrency Course

Building Programmable Money

From Bitcoin's Digital Gold to Ethereum's World Computer

Lesson Outline

1. Ethereum vs Bitcoin
2. Gas & Transactions
3. Solidity Basics
4. Deploy & Interact
5. Summary

Duration: 45 minutes — **Slides:** 30 — **Goal:** Understand Ethereum architecture, gas mechanics, and write your first smart contract

Section 1: Ethereum vs Bitcoin

8 minutes

Bitcoin: Digital Gold with Simple Scripts

Bitcoin's Design Philosophy

- **Purpose:** Store of value, peer-to-peer cash
- **Script:** Intentionally limited (non-Turing complete)
- **Operations:** Basic conditions for spending

Bitcoin Script Capabilities

- Multi-signature transactions
- Time-locked transactions (CLTV, CSV)
- Hash locks (HTLCs for Lightning)
- Simple conditional logic

Why Limited by Design?

+ Security through simplicity
+ Predictable execution
+ No infinite loops possible
+ Smaller attack surface

Limitations

- No loops or complex logic
- No state storage
- Limited programmability
- Can't build complex apps

Bitcoin Script: Stack-based, 100 opcodes, designed for transaction validation only

Ethereum: The World Computer

Ethereum's Vision (Vitalik Buterin, 2014)
A blockchain with a **Turing-complete** programming language enabling:

- Arbitrary computation on-chain
- Self-executing contracts
- Decentralized applications (dApps)
- Programmable money and assets

Key Innovation

- State stored on blockchain
- Code lives on-chain permanently
- Anyone can interact with contracts
- Trustless execution guaranteed

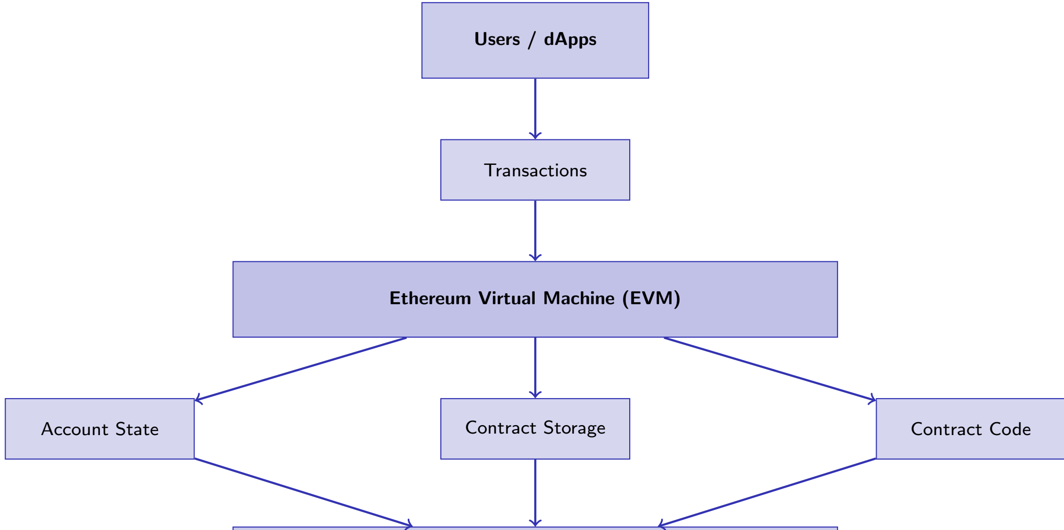
Comparison: Bitcoin vs Ethereum

Feature	BTC	ETH
Purpose	Currency	Platform
Scripts	Limited	Full
State	UTXO	Account
Loops	No	Yes
Storage	No	Yes
Gas	No	Yes

Result: DeFi, NFTs, DAOs, Tokens

Turing-complete: Can compute anything computable given enough resources

Ethereum Architecture Overview



Executes by

The Ethereum Virtual Machine (EVM)

What is the EVM?

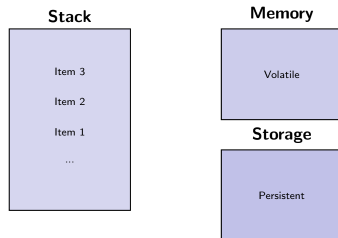
A **stack-based virtual machine** that:

- Executes smart contract bytecode
- Runs identically on every node
- Ensures deterministic results
- Meters computation with gas

EVM Characteristics

- 256-bit word size (for crypto ops)
- 1024 stack depth limit
- Memory: byte-addressable, volatile
- Storage: 256-bit to 256-bit mapping

EVM Execution Model



Opcodes: ADD, MUL, SLOAD, SSTORE, CALL, CREATE, SELFDESTRUCT, etc.

EVM bytecode is compiled from high-level languages like Solidity, Vyper, or Yul

Ethereum Accounts: EOA vs Contract

Externally Owned Accounts (EOA)
Controlled by private keys (humans/wallets):

- Has ETH balance
- Can send transactions
- No code associated
- Address from public key hash
- Signs transactions with private key

Account State

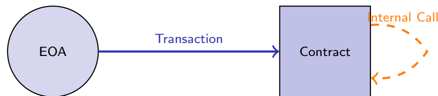
- `nonce`: Transaction count
- `balance`: ETH in Wei

Contract Accounts
Controlled by code (autonomous):

- Has ETH balance
- Has associated code
- Has persistent storage
- Address from creator + nonce
- Executes when triggered

Additional State

- `codeHash`: Hash of bytecode
- `storageRoot`: Merkle root of storage



Only EOAs can initiate transactions; contracts react to calls

Section 2: Gas & Transactions

7 minutes

What is Gas?

Gas = Computational Fuel

- Unit measuring computational work
- Every operation costs gas
- Paid in ETH (gas price x gas used)
- Prevents infinite loops
- Incentivizes efficient code

Why Not Just Use ETH?

- Decouples computation cost from ETH price
- Gas costs stay stable in gas units
- Price adjusts via market (gas price)

Gas Costs (Examples)

Operation	Gas
ADD/SUB	3
MUL/DIV	5
SLOAD (storage read)	2,100
SSTORE (new value)	20,000
SSTORE (update)	5,000
CREATE (contract)	32,000
Transaction base	21,000

Storage is expensive by design!

Gas prevents spam and DoS attacks; miners/validators prioritize higher-paying transactions

Gas Economics: Execution Cost

Transaction Fee Calculation

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

Example Transaction

- Gas Used: 50,000 gas
- Gas Price: 30 Gwei
- Fee: $50{,}000 \times 30 = 1{,}500{,}000$ Gwei
- Fee: 0.0015 ETH
- At $2000/ETH: $3.00

Unit Conversions

- 1 ETH = 10^{18} Wei
- 1 Gwei = 10^{9} Wei
- 1 ETH = 10^{9} Gwei

EIP-1559 Fee Structure (Post-London)

- **Base Fee:** Algorithmically set, burned
- **Priority Fee (Tip):** Goes to validator
- **Max Fee:** User's maximum willing to pay

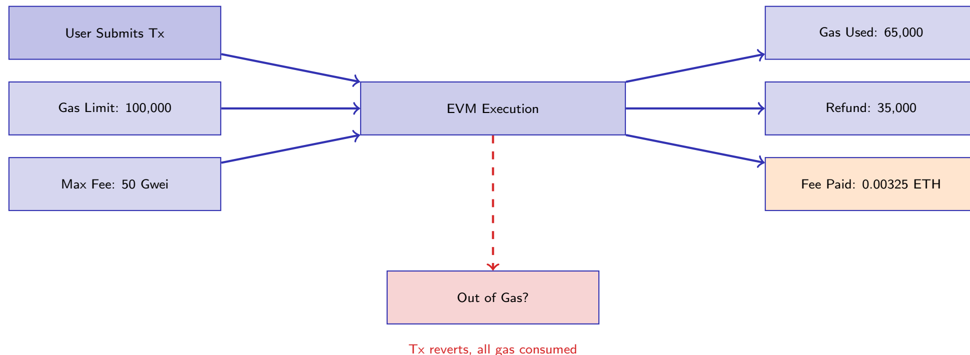
$$\text{Fee} = \text{Gas} \times (\text{Base} + \text{Tip})$$

Benefits of EIP-1559

- More predictable fees
- ETH becomes deflationary (burning)
- Better UX for fee estimation

Average transaction: 21,000 gas (transfer) to 500,000+ gas (complex contract interaction)

Gas Mechanics Visualization



Success Scenario

- Execution completes within limit
- Unused gas refunded to sender
- State changes persisted

Out of Gas Scenario

- Execution halts at limit
- ALL gas consumed (no refund)
- State changes reverted

Always estimate gas before sending; tools like Hardhat provide accurate estimates

Gas Components: Price, Limit, and Fees

Gas Limit

Maximum gas you authorize:

- Set by transaction sender
- Protects from runaway costs
- Too low = transaction fails
- Block gas limit: 30M

Tip: Use 20-50% buffer

Gas Price

How much you pay per gas unit:

- Market-driven (supply/demand)
- Higher = faster inclusion
- Measured in Gwei
- Fluctuates with network load

Check: etherscan.io/gastracker

Total Fee

What you actually pay:

- Gas Used $\times$ Effective Price
- Base fee burned (EIP-1559)
- Priority fee to validator
- Unused gas refunded

Max: Limit $\times$ Max Fee

$$\underbrace{\text{Gas Limit}}_{\text{You set}} \times \underbrace{(\text{Base Fee} + \text{Priority Fee})}_{\text{Network + You}} = \underbrace{\text{Max Possible Fee}}_{\text{Deducted}} \quad \rightarrow \quad \text{Refund: } (\textit{Limit} - \textit{Used}) \times \textit{Price}$$

Typical gas prices: 10-30 Gwei (low activity), 50-200 Gwei (high activity), 500+ Gwei (congestion)

Why Gas Prevents Infinite Loops

The Halting Problem

Can we determine if a program will finish?

- Mathematically proven impossible
- Turing-complete = can loop forever
- Ethereum allows Turing-completeness
- Solution: Economic constraint (gas)

Without Gas

- X Attacker deploys infinite loop
- X All nodes stuck forever
- X Network halts completely
- X DoS attack succeeds

With Gas

- ✓ Each operation costs gas
- ✓ Gas limit caps execution
- ✓ Loop runs until gas exhausted
- ✓ Attacker pays for each iteration

Example: Malicious Loop

```
// This would be expensive!
while(true) {
  storage[i] = i;  // 20,000 gas each
  i++;
}
// At 20,000 gas/iteration:
// 100,000 gas = 5 iterations only!
```

Gas makes computation economically bounded; attackers must pay proportionally to damage attempted

Section 3: Solidity Basics

15 minutes

What is Solidity?

Solidity Overview

- High-level language for smart contracts
- Influenced by C++, Python, JavaScript
- Compiles to EVM bytecode
- Statically typed
- Supports inheritance, libraries
- Most popular smart contract language

File Structure

- `.sol` file extension
- Pragma version declaration
- Import statements
- Contract definitions

Basic Contract Structure

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.20;
3
4  contract MyContract {
5      // State variables
6      uint256 public value;
7
8      // Constructor
9      constructor() {
10         value = 0;
11     }
12
13     // Functions
14     function setValue(uint256 _v) public {
15         value = _v;
16     }
17 }
```

Always specify SPDX license and pragma version; use latest stable (0.8.x) for security features

Solidity Data Types

Value Types

bool	true/false
uint	0 to $2^{256} - 1$
int	-2^{255} to $2^{255} - 1$
address	20-byte value
bytes32	Fixed bytes

Sized Variants

- uint8 to uint256
- int8 to int256
- bytes1 to bytes32

Reference Types

string	UTF-8 text
bytes	Dynamic bytes
array[]	Dynamic array
array[N]	Fixed array
struct	Custom type

Data Locations

- storage - persistent
- memory - temporary
- calldata - read-only

Special Types

mapping	Hash table
enum	Enumeration
contract	Contract ref

Mapping Example

```
// address -> balance
mapping(address => uint)
    public balances;

// Nested mapping
mapping(address =>
  mapping(address => uint))
    public allowance;
```

Mappings cannot be iterated; use arrays alongside if enumeration needed

State Variables

What Are State Variables?

- Permanently stored on blockchain
- Defined at contract level
- Persist between function calls
- Cost gas to read/write (storage)
- Form the contract's "memory"

Visibility Modifiers

- `public` - auto getter, accessible
- `private` - contract only
- `internal` - contract + derived

State Variable Examples

```
1  contract Bank {
2      // Simple state
3      address public owner;
4      uint256 public totalDeposits;
5
6      // Mapping state
7      mapping(address => uint256)
8          private balances;
9
10     // Array state
11     address[] public depositors;
12
13     // Struct state
14     struct Account {
15         uint256 balance;
16         bool active;
17     }
18     mapping(address => Account)
19         public accounts;
20 }
```

State variables live in storage slots; packing smaller types saves gas (e.g., uint128 + uint128 in one slot)

Functions and Visibility

Function Visibility

Modifier	Access
public	Anyone, internally
external	Only from outside
internal	Contract + children
private	Contract only

State Mutability

- view - reads state, no modify
- pure - no state access
- payable - can receive ETH
- (none) - can modify state

Function Syntax

```
contract Functions {
    uint256 public value;

    // State-changing
    function setValue(uint256 _v)
        public {
        value = _v;
    }

    // View function
    function getValue()
        public view returns (uint256) {
        return value;
    }

    // Pure function
    function add(uint a, uint b)
        public pure returns (uint) {
        return a + b;
    }
}
```

Use `external` over `public` when function only called externally (saves gas)

Events: Logging on the Blockchain

What Are Events?

- Logging mechanism for contracts
- Stored in transaction logs (not state)
- Cheaper than storage
- Can be filtered and searched
- Used by frontends/dApps

Use Cases

- Track token transfers
- Record important state changes
- Notify external applications
- Create audit trails

Event Syntax

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

    event Approval(
        address indexed owner,
        address indexed spender,
        uint256 value
    );

    function transfer(address to, uint v)
        public {
        // ... transfer logic ...
        emit Transfer(msg.sender, to, v);
    }
}
```

indexed: searchable (max 3)

Events cost 375 gas base + 375 per indexed topic + 8 per byte of data

Code Example: HelloWorld.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

/// @title HelloWorld
/// @notice First smart contract
contract HelloWorld {
    // State variable
    string public message;

    // Event for message changes
    event MessageChanged(
        address indexed sender,
        string newMessage
    );

    // Constructor sets initial msg
    constructor(string memory _msg) {
        message = _msg;
    }

    // Update the message
    function setMessage(string memory _m)
        public {
        message = _m;
        emit MessageChanged(msg.sender, _m);
    }
}
```

Code Walkthrough

1. **License & Pragma**: Required headers
2. **Contract**: Named container for code
3. **State Variable**: `message` stored on-chain
4. **Event**: Log when message changes
5. **Constructor**: Runs once at deployment
6. **Function**: Changes state, emits event

Key Observations

- `public` auto-creates getter
- `memory` for temp string storage
- `msg.sender` = caller address
- Constructor params set at deploy

This simple contract demonstrates state, events, constructor, and public functions

Code Example: SimpleStorage.sol

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.20;
3
4  /// @title SimpleStorage
5  /// @notice Store and retrieve values
6  contract SimpleStorage {
7      // State
8      uint256 private storedValue;
9      address public owner;
10
11     // Events
12     event ValueStored(uint256 indexed val);
13
14     // Constructor
15     constructor() {
16         owner = msg.sender;
17         storedValue = 0;
18     }
19
20     // Store a value
21     function store(uint256 _value) public {
22         storedValue = _value;
23         emit ValueStored(_value);
24     }
25
26     // Retrieve the value
27     function retrieve()
28         public view returns (uint256) {
29         return storedValue;
30     }
31 }
```

Key Concepts Demonstrated

- **Private vs Public**:
  - `storedValue` - no auto getter
  - `owner` - has auto getter
- **Explicit Getter**: `retrieve()` function
- **Owner Pattern**: Track deployer

Gas Costs

- `store()`: 43,000 gas (first write)
- `store()`: 26,000 gas (update)
- `retrieve()`: FREE (view)

Why Free?

View functions execute locally without transaction (no state change, no gas).

Pattern: Track owner in constructor for access control; retrieve() is a view (free)

Constructor and Deployment

Constructor Characteristics

- Runs **once** at deployment
- Cannot be called again
- Initializes state variables
- Can receive parameters
- Can receive ETH (`payable`)

Constructor Patterns

```
constructor() {
    owner = msg.sender;  // Track deployer
}

constructor(uint256 _initial) {
    value = _initial;  // Parameterized
}

constructor() payable {
    // Can receive ETH at deploy
}
```

Deployment Process

1. Compile Solidity to bytecode
2. Create deployment transaction
3. Include constructor args (ABI-encoded)
4. Send to null address (0x0)
5. EVM executes init code
6. Contract gets address
7. Runtime bytecode stored on-chain

Contract Address

$$\text{address} = \text{keccak256}(\text{sender}, \text{nonce})$$

Predictable! CREATE2 allows salt-based addresses.

Deployment gas = base cost (32,000) + init code execution + runtime code storage (200 gas/byte)

View vs Pure Functions

View Functions

Read state but don't modify:

```
uint256 public balance;

// Can read state
function getBalance()
    public view returns (uint256) {
    return balance;  // OK: reading
}

function getDoubleBalance()
    public view returns (uint256) {
    return balance * 2;  // OK
}

// ERROR: view cannot modify
function setBalance(uint _b)
    public view {
    balance = _b;  // COMPILE ERROR
}
```

Pure Functions

No state access at all:

```
// Pure: only uses parameters
function add(uint a, uint b)
    public pure returns (uint) {
    return a + b;  // OK
}

function hash(bytes memory data)
    public pure returns (bytes32) {
    return keccak256(data);  // OK
}

// ERROR: pure cannot read state
function pureError()
    public pure returns (uint) {
    return balance;  // COMPILE ERROR
}
```

	Read State	Modify State	Cost (external call)
`view`	Yes	No	Free
`pure`	No	No	Free
(none)	Yes	Yes	Gas

Always use view/pure when possible; the compiler enforces these constraints

Modifiers: Reusable Function Guards

What Are Modifiers?

- Reusable code for function checks
- Execute before/after function body
- Common for access control
- Use _ as function placeholder

Modifier Syntax

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

modifier nonReentrant() {
    require(!locked, "Reentrant");
    locked = true;
    _;  // Function executes
    locked = false;  // After function
}
```

Using Modifiers

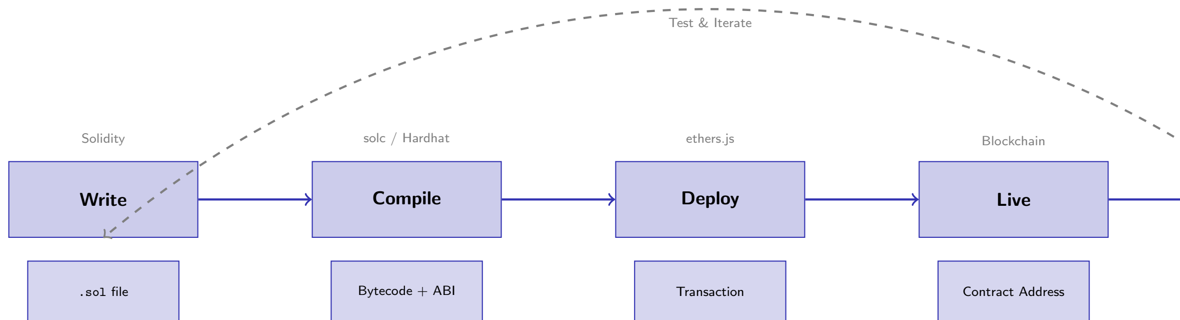
```
contract Ownable {
    address public owner;
    bool private locked;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }

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

    // Multiple modifiers
    function sensitive()
        public onlyOwner nonReentrant {
        // Protected code
    }
}
```

Modifiers reduce code duplication; OpenZeppelin provides battle-tested modifiers

Section 4: Deploy & Interact

10 minutes

Smart Contract Lifecycle



Development

- Write Solidity code
- Compile to bytecode
- Generate ABI (interface)

Deployment

- Send creation tx
- Pay deployment gas
- Get contract address

Interaction

- Call view functions (free)
- Send transactions (gas)
- Emit/listen to events

Development Tools: Hardhat

What is Hardhat?

Modern Ethereum development environment:

- Local blockchain network
- Compilation and testing
- Deployment scripts
- Debugging with stack traces
- Plugin ecosystem

Project Setup

```
# Initialize project
npm init -y
npm install --save-dev hardhat

# Create Hardhat project
npx hardhat init

# Project structure:
# contracts/   <- Solidity files
# scripts/     <- Deploy scripts
# test/        <- Test files
# hardhat.config.js
```

Common Commands

```
# Compile contracts
npx hardhat compile

# Run local blockchain
npx hardhat node

# Run tests
npx hardhat test

# Deploy to local network
npx hardhat run scripts/deploy.js
  --network localhost

# Deploy to testnet
npx hardhat run scripts/deploy.js
  --network sepolia
```

Alternatives

- Foundry (Rust-based, fast)
- Remix (browser IDE)
- Truffle (legacy)

Hardhat provides console.log() in Solidity for debugging - invaluable during development

Local Blockchain Setup

Hardhat Network

Built-in local Ethereum network:

- Instant mining (no wait)
- Pre-funded test accounts
- Console.log support
- Time manipulation
- State snapshots

Start Local Node

```
$ npx hardhat node

Started HTTP and WebSocket JSON-RPC
server at http://127.0.0.1:8545/

Account #0: 0xf39Fd6e51aad88F...
Private Key: 0xac0974bec39a17e...
(10000 ETH)

Account #1: 0x70997970C51812d...
Private Key: 0x59c6995e998f97a...
(10000 ETH)
...
```

Network Configuration

```
// hardhat.config.js
module.exports = {
  solidity: "0.8.20",
  networks: {
    localhost: {
      url: "http://127.0.0.1:8545"
    },
    sepolia: {
      url: process.env.SEPOLIA_URL,
      accounts: [process.env.PRIVATE_KEY]
    },
    mainnet: {
      url: process.env.MAINNET_URL,
      accounts: [process.env.PRIVATE_KEY]
    }
  }
};
```

Test Networks

- Sepolia - Ethereum testnet
- Goerli - Being deprecated
- Holesky - New testnet

Never use mainnet private keys in config files; use environment variables

Code: Deployment Script

scripts/deploy.js

```javascript
const { ethers } = require("hardhat");

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

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

  // Get contract factory
  const SimpleStorage = await
    ethers.getContractFactory(
      "SimpleStorage"
    );

  // Deploy contract
  const storage = await
    SimpleStorage.deploy();

  // Wait for deployment
  await storage.waitForDeployment();

  console.log("Deployed to:",
    await storage.getAddress());
}

main().catch(console.error);
```

Deployment with Constructor Args

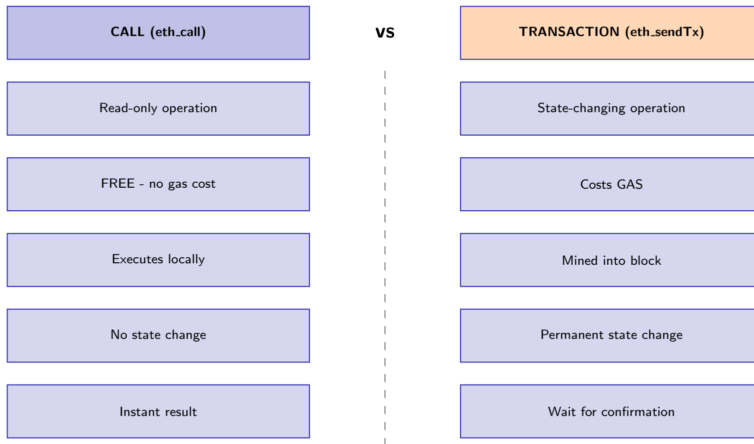
```javascript
async function main() {
  const HelloWorld = await
    ethers.getContractFactory(
      "HelloWorld"
    );

  // Pass constructor argument
  const hello = await
    HelloWorld.deploy("Hello!");

  await hello.waitForDeployment();

  // Verify initial state
  const msg = await hello.message();
  console.log("Message:", msg);
}
```

Run Deployment

```
$ npx hardhat run scripts/deploy.js
Deploying with: 0xf39Fd6e51aad...
Deployed to: 0x5FbDB2315678...
```

ethers.js v6 uses getContractFactory() and waitForDeployment(); v5 used deployed()

Call vs Transaction



Use CALL for:

- `view` functions
- `pure` functions

Use TRANSACTION for:

- Modifying state variables
- Transferring ETH/tokens

Code: Python web3 Interaction

Reading from Contract

```python
from web3 import Web3

# Connect to local node
w3 = Web3(Web3.HTTPProvider(
    'http://localhost:8545'
))

# Contract ABI (simplified)
abi = [
  {"name": "retrieve",
   "type": "function",
   "outputs": [{"type": "uint256"}],
   "stateMutability": "view"},
  {"name": "store",
   "type": "function",
   "inputs": [{"type": "uint256"}]}
]

# Contract instance
contract = w3.eth.contract(
    address='0x5FbDB2315678...',
    abi=abi
)

# Call view function (free)
value = contract.functions.retrieve()
    .call()
print(f"Stored value: {value}")
```

Writing to Contract

```python
# Your account
account = '0xf39Fd6e51aad88F...'
private_key = '0xac0974bec39a...'

# Build transaction
tx = contract.functions.store(42)
    .build_transaction({
        'from': account,
        'nonce': w3.eth.get_
            transaction_count(account),
        'gas': 100000,
        'gasPrice': w3.to_wei(
            '30', 'gwei')
    })

# Sign transaction
signed = w3.eth.account
    .sign_transaction(
        tx, private_key
    )

# Send and wait
tx_hash = w3.eth
    .send_raw_transaction(
        signed.raw_transaction
    )
receipt = w3.eth
    .wait_for_transaction_receipt(
        tx_hash
    )
print(f"Gas used: {receipt.gasUsed}")
```

Testing Your Contract

test/SimpleStorage.test.js

```
const { expect } = require("chai");
const { ethers } = require("hardhat");

describe("SimpleStorage", function() {
  let storage;
  let owner;

  beforeEach(async function() {
    [owner] = await ethers.getSigners();
    const Factory = await ethers
      .getContractFactory(
        "SimpleStorage"
      );
    storage = await Factory.deploy();
  });

  it("should store value",
    async function() {
    await storage.store(42);
    expect(await storage.retrieve())
      .to.equal(42n);
  });

  it("should emit event",
    async function() {
    await expect(storage.store(100))
      .to.emit(storage, "ValueStored")
      .withArgs(100);
  });
});
```

Run Tests

```
$ npx hardhat test

  SimpleStorage
    V should store value (45ms)
    V should emit event (38ms)

  2 passing (1s)
```

Testing Best Practices

- Test all public functions
- Test edge cases (0, max values)
- Test access control (modifiers)
- Test events are emitted
- Test reverts with `revertedWith`
- Use `beforeEach` for fresh state

Coverage

```
$ npx hardhat coverage
# Generates coverage report
```

Testing is crucial: bugs in deployed contracts can mean lost funds with no recourse

Section 5: Summary

5 minutes

Key Takeaways

Ethereum Fundamentals

- Ethereum = programmable blockchain
- EVM executes smart contracts
- Two account types: EOA and Contract
- State stored on-chain globally

Gas Mechanics

- Gas measures computation cost
- Prevents infinite loops
- Fee = Gas Used $\times$ Gas Price
- Storage operations are expensive

Solidity Essentials

- State variables = persistent storage
- Functions: public, external, view, pure
- Events for logging (cheaper than storage)
- Modifiers for access control

Development Workflow

- Write $\rightarrow$ Compile $\rightarrow$ Deploy $\rightarrow$ Interact
- Use Hardhat for local development
- Test thoroughly before mainnet
- Call (free) vs Transaction (gas)

Golden Rule: Smart contracts are immutable. Test extensively, audit code, and start with testnets before touching real ETH.

You now have the foundation to write and deploy your own smart contracts!

Next Lesson: Create Your Own Token

Lesson 4 Preview

- ERC-20 Token Standard
- Token interface and functions
- Build a complete token contract
- Deploy your own cryptocurrency
- Transfer, approve, allowance pattern
- Real-world token use cases

What You'll Build

- Your own ERC-20 token
- Token with custom features
- Integration with DeFi concepts

Preparation

1. Review today's Solidity basics
2. Practice with SimpleStorage
3. Set up Hardhat environment
4. Get Sepolia testnet ETH

Resources

- Solidity docs: docs.soliditylang.org
- OpenZeppelin: openzeppelin.com/contracts
- Hardhat: hardhat.org
- Etherscan: etherscan.io

Faucet: sepoliafaucet.com

Lesson 4: From smart contracts to real tokens - the foundation of DeFi and NFTs

Questions?

Thank you for your attention