Solidity Workshop: A Teaching Walk-through

What you will build, how the EVM runs it, what to look for

Cryptoeconomics Course

Prof. Joerg Osterrieder

Spring 2026

What the notebook builds

The notebook compiles real Solidity 0.8.20 and deploys it to an in-process EVM running entirely inside the Colab runtime. Three contracts, in order:

1. **SimpleStorage**: one number on chain. Teaches state variables, `public`, and `view` functions.
2. **MyToken (ERC-20)**: built incrementally in 5 sub-steps. Teaches `mapping`, `require`, events, and the `approve` / `transferFrom` delegation pattern.
3. **SimpleVoting**: a small DAO. Teaches `struct`, dynamic arrays, and access control via the `onlyOwner` `modifier`.

Tooling. `py-solc-x` (compiles Solidity 0.8.20), `web3.py` (sends transactions), `eth-tester[py-evm]` (the in-process EVM with 10 pre-funded accounts).

Three contracts, one notebook, one Python kernel. No wallet, no faucet, no testnet.

SimpleStorage: state variable, setter, view function

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

contract SimpleStorage {
    uint256 public storedNumber;

    function set(uint256 _num) public {
        storedNumber = _num;
    }

    function get() public view returns (uint256) {
        return storedNumber;
    }
}
```

Three concepts on this slide.

- `uint256 public storedNumber` declares a 256-bit unsigned integer in contract *storage*; `public` auto-generates a getter so external callers can read it.
- `set` is a state-changing transaction. The EVM charges gas (about 22,000 on the first write, about 5,000 to update an existing slot).
- `view` promises the function does not modify state. Calling a `view` function from outside is free; no transaction is sent.

Read is free, write is paid. Every contract in this deck is a variation on that one rule.

The notebook's compile-deploy-call loop

Every contract in the notebook is driven from Python with the same three steps:

```
# 1. Compile
compiled = solcx.compile_source(src, solc_version='0.8.20')
abi      = compiled['<stdin>:SimpleStorage']['abi']
bytecode = compiled['<stdin>:SimpleStorage']['bin']

# 2. Deploy
SimpleStorage = w3.eth.contract(abi=abi, bytecode=bytecode)
tx = SimpleStorage.constructor().transact({'from': deployer})
addr = w3.eth.wait_for_transaction_receipt(tx).contractAddress
simple = w3.eth.contract(address=addr, abi=abi)

# 3. Call (write -> .transact, view -> .call)
simple.functions.set(42).transact({'from': deployer})
value = simple.functions.get().call()
```

Watch the receipt. `gasUsed` is the actual cost. `logs` contains the events the contract emitted. If `require` fails, `transact` raises a Python exception you can catch.

Same three steps for SimpleStorage, MyToken, and SimpleVoting. Memorise this loop.

MyToken (ERC-20): storage layout and constructor

```
contract MyToken {
    string public name = "MyToken";
    string public symbol = "MTK";
    uint8 public decimals = 18;
    uint256 public totalSupply;

    mapping(address => uint256) public balanceOf;

    constructor(uint256 _initialSupply) {
        totalSupply = _initialSupply;
        balanceOf[msg.sender] = _initialSupply;
    }
}
```

Three concepts on this slide.

- `mapping(address => uint256)` is a hash table that lives in storage. Every key starts at the zero value (0) without being initialised. You cannot iterate it.
- `msg.sender` is the address that called the current function. Inside the constructor, that is the deployer; the constructor mints the entire initial supply to that address.
- `public` on `balanceOf` auto-generates the standard ERC-20 getter `balanceOf(address) returns (uint256)`.

State only. No behaviour yet. The next slide adds the function that actually moves tokens.

MyToken: transfer, require, and the Transfer event

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

function transfer(address to, uint256 amount) public returns (bool) {
    require(balanceOf[msg.sender] >= amount, "Insufficient balance");
    balanceOf[msg.sender] -= amount;
    balanceOf[to]         += amount;
    emit Transfer(msg.sender, to, amount);
    return true;
}
```

Three concepts on this slide.

- `require(cond, "msg")` reverts the entire transaction if the condition is false. All state changes in this transaction are rolled back; remaining gas is refunded.
- *Checks-Effects-Interactions:* validate inputs first, mutate state second, call out to other contracts last. Following this order prevents most reentrancy bugs.
- `event Transfer(...)` declares a log entry; `emit` writes one. `indexed` fields are searchable by wallets and explorers without reading storage.

Wallets do not poll storage. They listen for Transfer events. Without emit, your token is invisible.

MyToken: approve and transferFrom (delegation pattern)

```
mapping(address => mapping(address => uint256)) public allowance;
event Approval(address indexed owner, address indexed spender, uint256 value);

function approve(address spender, uint256 amount) public returns (bool) {
    allowance[msg.sender][spender] = amount;
    emit Approval(msg.sender, spender, amount);
    return true;
}

function transferFrom(address from, address to, uint256 amount) public returns (bool) {
    require(balanceOf[from] >= amount, "Insufficient balance");
    require(allowance[from][msg.sender] >= amount, "Allowance exceeded");
    balanceOf[from] -= amount;
    balanceOf[to]   += amount;
    allowance[from][msg.sender] -= amount;
    emit Transfer(from, to, amount);
    return true;
}
```

Why two functions. `approve` sets a per-spender cap; `transferFrom` lets the spender pull within that cap. This is how every DEX, lending protocol, and NFT marketplace moves tokens with the owner's consent.

The two-step dance you see in every Web3 sign-in: 1. approve a contract, 2. let the contract pull.

SimpleVoting: struct, modifier, access control

```
contract SimpleVoting {
    struct Proposal { string description; uint256 voteCount; }
    Proposal[] public proposals;
    mapping(address => bool) public hasVoted;
    address public owner;

    modifier onlyOwner() { require(msg.sender == owner, "not owner"); _; }

    constructor() { owner = msg.sender; }

    function createProposal(string memory _desc) public onlyOwner {
        proposals.push(Proposal({description: _desc, voteCount: 0}));
    }

    function vote(uint256 proposalId) public {
        require(!hasVoted[msg.sender], "already voted");
        hasVoted[msg.sender] = true;
        proposals[proposalId].voteCount += 1;
    }
}
```

Concepts. `struct` groups fields into a record. `Proposal[]` is a dynamic storage array. `modifier` runs its body before `_;`, where the function body splices in. `string memory` is required because reference-type parameters must declare their data location.

Write the access check once, attach it to many functions.

What to watch when you run the notebook

- **Gas costs.** The notebook prints `gasUsed` after every state-changing call. Compare `set(42)` on a fresh slot (∼22,000 gas) to `set(99)` on the same slot (∼5,000 gas). The first write pays for slot allocation; updates do not.
- **The deliberate revert.** In Section 3, the notebook calls `transferFrom(bob, carol, 10)` without an approval. The second `require` fails; the transaction reverts; Python catches the exception and prints `caught expected revert: ...`.
- **The `onlyOwner` guard.** In Section 4, a non-owner calls `createProposal`. The modifier rejects it. The cell prints `non-owner correctly rejected: ...`.
- **Events in the receipt.** After each `transfer`, the receipt's `logs` array contains the `Transfer` event. This is what wallets index to update your balance display.
- **The `view` contrast.** `balanceOf(addr).call()` costs nothing because it does not produce a transaction; `transfer(...).transact()` does.

Five things to spot. Open the notebook, run all, and tick them off.

Open the notebook

Run it now. The first cell takes about 30 seconds (downloading `solc 0.8.20`); every cell after that runs in milliseconds.

Open in Colab | https://colab.research.google.com/github/Digital-AI-Finance/crypto-economics/blob/main/10_solidity_basics/10_solidity_workshop.ipynb

Three exercises at the end of the notebook.

1. Add a `burn(uint256 amount)` function to `MyToken`: subtract from `balanceOf[msg.sender]`, decrement `totalSupply`, emit `Transfer(msg.sender, address(0), amount)`.
2. Cap the supply at 10 million with a `MAX_SUPPLY` constant and an owner-only `mint(address, uint256)` that requires `totalSupply + amount <= MAX_SUPPLY`.
3. Make voting weighted by token balance: take `MyToken` as a constructor argument; in `vote`, increment `voteCount` by `token.balanceOf(msg.sender)` instead of by 1.

For the formal version of all this material, see the full lecture in `10_solidity.pdf`.

Done in the next 30 minutes: deploy three contracts, send tokens, count votes, plot the tally.