Solidity Basics

Lesson 10: Writing Your First Smart Contracts

Prof. Joerg Osterrieder

Spring 2026

Learning Objectives

After this lesson, you will be able to:

1. Write and understand basic Solidity smart contracts
2. Deploy and interact with contracts using Remix IDE
3. Build an ERC-20 token from scratch
4. Implement access control with modifiers

Prerequisites: Lessons 1–7 (especially L07: Smart Contracts and Game Theory)

taught what smart contracts do — this lesson teaches how to write them

Lesson Outline

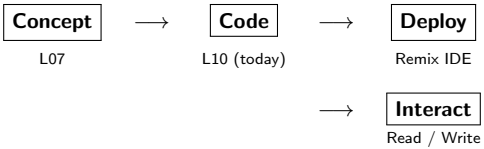
1. What is Solidity?
2. Your First Contract
3. Building an ERC-20 Token
4. Building a Voting Contract
5. Deploying with Remix

What is Solidity?

Where We Are

Lesson 7 — Concepts: What smart contracts *do* (trustless execution, token standards, security, game theory).

Lesson 10 — Code: How to *write* smart contracts in Solidity.



Our path today:

- Learn Solidity syntax from zero
- Build a complete ERC-20 token step by step
- Build a voting contract with access control
- Deploy everything on Remix IDE

No prior programming knowledge assumed — every keyword is explained on first use

The Solidity Language

History: Created by Gavin Wood in 2014 specifically for Ethereum smart contracts.

Key characteristics:

- **Statically typed** — every variable must declare its type (number, address, etc.)
- Compiles to **bytecode** — machine code that runs on the **EVM** (Ethereum Virtual Machine)
- Syntax inspired by JavaScript, C++, and Python

Feature	Solidity	JavaScript	Python
Typed variables	Yes (static)	No (dynamic)	No (dynamic)
Semicolons	Required	Optional	No
Curly braces	Yes	Yes	No (indentation)
Runs on	EVM	Browser/Node	OS

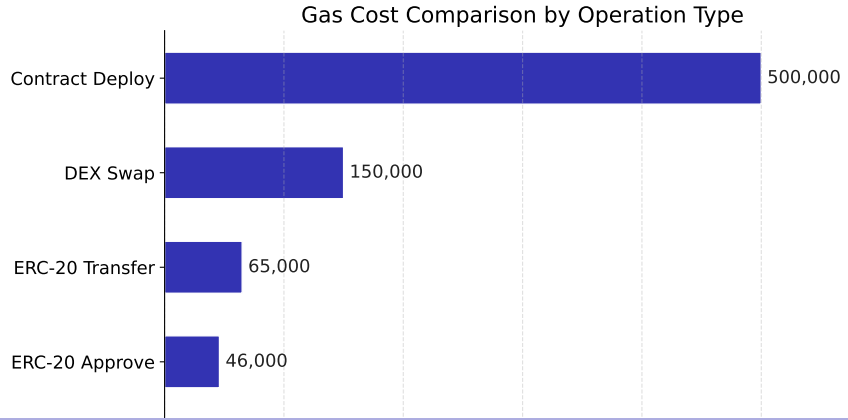
Bytecode: The compiled, low-level instructions that the EVM executes.
EVM: The virtual computer inside every Ethereum node that runs contract bytecode.

Solidi

is the most widely used smart contract language — over 80% of Ethereum contracts

How Smart Contracts Run

Gas (revisited from L07): Every operation costs **gas** — a fee paid in ETH to compensate validators.



Data Types in Solidity

Every variable in Solidity must have a declared type. Here are the most common:

Type	Description	Example Value
`uint` / `uint256`	Unsigned integer (0 and up)	`42`
`int` / `int256`	Signed integer (negative allowed)	`-7`
`bool`	True or false	`true`
`address`	20-byte Ethereum address	`0xAb5...3f`
`string`	Text (UTF-8)	`"Hello"`
`bytes`	Raw byte data	`0xca fe`
`T[]`	Dynamic array of type T	`[1, 2, 3]`
`mapping`	Key–value lookup table	*(covered in Slide 18)*

- `uint` is short for `uint256` — a 256-bit unsigned integer (range: 0 to $2^{256} - 1$)
- `address` holds any Ethereum account or contract address (20 bytes = 40 hex characters)
- `mapping` is Solidity's most powerful data structure — we explain it fully when we build our ERC-20 token

Solidi

has no floating-point numbers — all amounts are integers (e.g. wei, the smallest ETH unit)

Your First Contract

Anatomy of a Contract

The simplest possible Solidity file:

```
1 // SPDX-License-Identifier: MIT          ← License identifier (required by compiler)
2 pragma solidity ^0.8.20;                 ← pragma: tells compiler which version to use
3
4 contract SimpleStorage {                 ← contract: defines a new smart contract
5                                          ← (empty body --- we will add code next)
6 }                                        ← closing brace ends the contract
```

Two new keywords:

- `pragma solidity ^0.8.20;` — tells the compiler: "use version 0.8.20 or compatible." Every Solidity file starts with a pragma.
- `contract` — like a "class" in other languages. It groups related data and functions that live on the blockchain.

This contract compiles but does nothing yet. Next, we add *state variables* and *functions*.

The

^ symbol means "compatible with" — ^0.8.20 accepts 0.8.20, 0.8.21, etc. but not 0.9.0

State Variables

Add a variable that *lives on the blockchain*:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.20;
3
4 contract SimpleStorage {
5     uint256 public storedNumber;          ← state variable: stored on the blockchain
6 }                                         ← public: anyone can read this value
```

What is a state variable?

- A variable declared inside a contract but outside any function
- Its value is permanently stored on the blockchain (in "storage")
- **Writing** to it costs gas (changes the blockchain state)
- **Reading** it is free when called from outside (no transaction needed)

New keyword: `public` — a *visibility modifier*. It means:

- Anyone (any user or contract) can read this variable
- Solidity automatically creates a "getter" function for it

uint2

is the full name for `uint` — both mean a 256-bit unsigned integer

Functions: Write

Add a function that *changes* the stored value:

```
contract SimpleStorage {
    uint256 public storedNumber;

    function set(uint256 _num) public {    ← function:  a named block of code
                                           ← _num is a parameter (input value)
        storedNumber = _num;               ← assigns the input to the state variable
    }
}
```

New keywords:

- `function` — defines a reusable block of code. Syntax: `function name(inputs) visibility { body }`
- **Parameter** (`_num`) — an input value passed by the caller. The underscore prefix is a Solidity convention for parameters.

What happens when someone calls `set(42)`?

1. A transaction is sent to the blockchain
2. The EVM executes `storedNumber = 42`
3. The caller pays gas for the storage write (∼20,000 gas)

that change state require a transaction and cost gas

Functions: Read

Add a function that *reads* without changing state:

```
contract SimpleStorage {
    uint256 public storedNumber;

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

    function get() public view returns (uint256) {  ← view:  promises not to modify state
                                                     ← returns:  declares the output type
        return storedNumber;                         ← sends the value back to the caller
    }
}
```

New keywords:

- `view` — this function only *reads* data; it never writes to storage. Calling a `view` function is **free** (no gas, no transaction).
- `returns (uint256)` — declares what type the function sends back.
- `pure` — even stricter than `view`: the function does not read *or* write state. Example: a math helper like `function add(uint a, uint b) pure returns (uint) { return a + b; }`.

Summary: `view` = reads state, `pure` = reads nothing, no modifier = may write state.

The

`public` **keyword on** `storedNumber` **already auto-generates a getter, but writing** `get()` **explicitly helps with clarity**

The Complete SimpleStorage

Our first working contract — 10 lines of Solidity:

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.20;                    ← 1. Version pragma
3
4  contract SimpleStorage {                    ← 2. Contract declaration
5      uint256 public storedNumber;            ← 3. State variable
6
7      function set(uint256 _num) public {     ← 4. Write function
8          storedNumber = _num;
9      }
10
11     function get() public view returns (uint256) {  ← 5. Read function (view)
12         return storedNumber;
13     }
14 }
```

What we have built:

- A contract that stores one number on the Ethereum blockchain
- Anyone can write a new number (`set`) — costs gas
- Anyone can read the current number (`get`) — free

Next step: deploy this on Remix IDE and interact with it live.

This pattern — state variable + write function + read function — is the foundation of every smart contract

Remix IDE: Compile + Deploy

Remix (`https://remix.ethereum.org`) is a free, browser-based IDE for Solidity.

Remix IDE Development Workflow

Step-by-step:

1. **Create file:** Click "New File" → name it `SimpleStorage.sol`
2. **Write code:** Paste the SimpleStorage contract
3. **Compile:** Click the Solidity compiler tab → select version `0.8.20` → click "Compile"
4. **Deploy:** Click "Deploy & Run" tab → select "Remix VM" (local test blockchain) → click "Deploy"
5. **Interact:** Expand the deployed contract → click `set` (enter a number) → click `get` (read it back)

VM is a simulated blockchain in your browser — no real ETH needed, instant transactions

Building an ERC-20 Token

What is ERC-20?

Recap from L07: ERC-20 is a *standard interface* that every fungible token on Ethereum must follow so that wallets, exchanges, and other contracts can interact with it.

Required functions (we will build all of them):

Function	Purpose
`totalSupply()`	How many tokens exist
`balanceOf(addr)`	How many tokens an address holds
`transfer(to, amount)`	Send your tokens to someone
`approve(spender, amount)`	Allow another address to spend your tokens
`allowance(owner, spender)`	Check how much a spender is approved for
`transferFrom(from, to, amount)`	Spend tokens on behalf of another address

Why build from scratch?

- Understand exactly how tokens work under the hood
- Production tokens typically use audited libraries, but learning means building

ERC

= Ethereum Request for Comments — ERC-20 was proposed in 2015 and is the most widely used token standard

Step 1: Contract Shell + Mappings

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

contract MyToken {
    string public name = "MyToken";        ← token name (human-readable)
    string public symbol = "MTK";          ← ticker symbol (like ETH, BTC)
    uint8 public decimals = 18;            ← divisibility (18 = standard)
    uint256 public totalSupply;            ← total tokens in existence

    mapping(address => uint256) public balanceOf;  ← mapping:  who owns how many tokens
}
```

New concept — `mapping`:

- A **key–value lookup table**, like a dictionary or hash map
- `mapping(address => uint256)` means: "given an address, return a number"
- Every address starts with a default value of `0`
- You *cannot* iterate over a mapping (no "list all keys")
- Lookups are $O(1)$ — constant time, very gas-efficient

Example: `balanceOf[0xAb5...3f]` → `1000` means that address holds 1,000 tokens.

mappi

is Solidity's most important data structure — it stores token balances, allowances, votes, and more

Step 2: 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) {   ← constructor:  runs once at deployment
        totalSupply = _initialSupply;       ← set total supply
        balanceOf[msg.sender] = _initialSupply;   ← give ALL tokens to deployer
    }                                       ← msg.sender:  the address that called this
}
```

New keywords:

- `constructor` — a special function that runs **exactly once** when the contract is deployed. It sets up the initial state. You cannot call it again after deployment.
- `msg.sender` — a built-in global variable that always contains the address of whoever is calling the current function. In the constructor, it is the deployer's address.

What happens at deployment:

1. Deployer calls `new MyToken(1000000)`
2. Constructor sets `totalSupply = 1000000`
3. All 1,000,000 tokens go to the deployer's balance

msg.s

changes depending on who calls the function — it is the caller's address, not the contract's

Step 3: Transfer Function

```
function transfer(address _to, uint256 _amount)
    public returns (bool)
{
    require(balanceOf[msg.sender] >= _amount,   ← require: check condition or revert
            "Insufficient balance");            ← error message if check fails

    balanceOf[msg.sender] -= _amount;           ← subtract from sender
    balanceOf[_to] += _amount;                  ← add to recipient

    return true;                                ← return success (ERC-20 spec)
}
```

New keyword — `require`:

- `require(condition, "error message")` — if the condition is `false`, the entire transaction **reverts** (all changes are undone, remaining gas is refunded).
- This is Solidity's primary input validation tool.

The Checks-Effects-Interactions pattern:

1. **Check:** Verify conditions (`require`)
2. **Effect:** Update state (subtract/add balances)
3. **Interact:** Call external contracts (none here)

Alway

check conditions BEFORE changing state — this prevents reentrancy attacks (covered in L07)

Step 4: Events

```
event Transfer(                              ← event:  declares a log entry
    address indexed from,                    ← indexed:  can be searched/filtered
    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);  ← emit:  write the event to the log

    return true;
}
```

New keywords:

- `event` — declares a log entry structure. Events are stored in transaction logs (cheap, not in contract storage).
- `emit` — writes an event to the blockchain log. Wallets and apps listen for these.
- `indexed` — marks a field as searchable. You can filter logs by indexed fields (e.g., "show all transfers to address X").

Event

are how wallets know your balance changed — they are 10x cheaper than storage writes

Step 5: Approve + Allowance

```
mapping(address => mapping(address => uint256))  ← nested mapping:  owner → spender → amount
    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;   ← set how much _spender can use
    emit Approval(msg.sender, _spender, _amount);
    return true;
}
```

Nested mapping — `mapping(address => mapping(address => uint256))`:

- First key: the token *owner*
- Second key: the approved *spender*
- Value: how many tokens the spender may use

Why do we need `approve`**?** It enables smart contracts (like DEXes) to move tokens on your behalf — you approve, then the DEX calls `transferFrom`.

The approve + transferFrom pattern is how every DEX, lending protocol, and DeFi app moves your tokens

Step 6: TransferFrom

```
function transferFrom(
    address _from,
    address _to,
    uint256 _amount
) public returns (bool) {
    require(balanceOf[_from] >= _amount,         ← check: sender has enough tokens
            "Insufficient balance");
    require(allowance[_from][msg.sender] >= _amount,  ← check: caller is approved
            "Allowance exceeded");

    balanceOf[_from] -= _amount;                 ← subtract from owner
    balanceOf[_to] += _amount;                   ← add to recipient
    allowance[_from][msg.sender] -= _amount;     ← decrease remaining allowance

    emit Transfer(_from, _to, _amount);
    return true;
}
```

How `transferFrom` works:

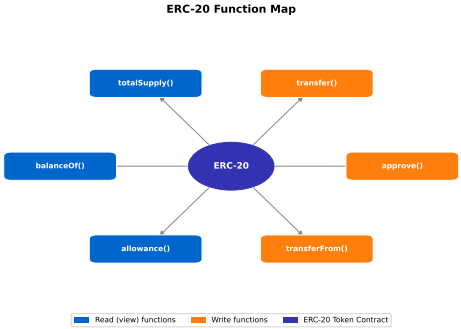
1. Alice calls `approve(DEX_address, 100)` — "DEX may spend 100 of my tokens"
2. DEX calls `transferFrom(Alice, DEX, 100)` — moves Alice's tokens
3. Allowance decreases from 100 to 0

Two `require` checks — both must pass or the transaction reverts.

This completes the ERC-20 interface — our token is now compatible with any Ethereum wallet or exchange

The Complete ERC-20 Token

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

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

    mapping(address => uint256) public balanceOf;
    mapping(address => mapping(address => uint256))
        public allowance;

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

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

    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;
    }

    function approve(address _spender, uint256 _amount)
```



ERC-20 Checklist:

- ✓ totalSupply()
- ✓ balanceOf()
- ✓ transfer()
- ✓ approve()
- ✓ allowance()
- ✓ transferFrom()

Deploy Your Token on Remix

Deploy and test your ERC-20 token in 5 steps:

1. **Create file:** In Remix, create `MyToken.sol` and paste the complete contract
2. **Compile:** Select Solidity `0.8.20` → click "Compile MyToken.sol"
3. **Deploy:** In "Deploy & Run", enter the initial supply (e.g., `1000000`) next to the Deploy button → click "Deploy"
4. **Check balance:** Expand the deployed contract → paste your address into `balanceOf` → click — you should see your full supply
5. **Transfer tokens:** Enter a different test address and an amount into `transfer` → click → verify both balances changed

Testing the approve/transferFrom flow:

1. From Account A, call `approve(Account_B, 500)`
2. Switch to Account B in Remix's account dropdown
3. Call `transferFrom(Account_A, Account_B, 500)`
4. Verify: Account A lost 500 tokens, Account B gained 500, allowance is now 0

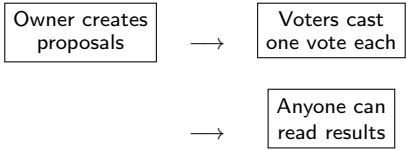
Tip: Remix provides multiple test accounts with 100 ETH each — use the dropdown to switch between them.

VM transactions are instant and free — experiment as much as you want

Building a Voting Contract

Voting Contract Goals

What we want to build (connecting to L09: Voting Systems):



Requirements:

- Only the contract owner can create proposals
- Each address can vote exactly once (no double voting)
- Votes are recorded permanently on the blockchain
- An event is emitted for every vote cast
- Anyone can query the current winner

New concepts we will learn:

- `struct` — custom data types
- `modifier` — reusable access control
- `memory` vs `storage` — where data lives

This voting contract demonstrates access control — a fundamental pattern in every smart contract

Step 1: State + Structs

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.20;
3
4  contract SimpleVoting {
5      struct Proposal {                        ← struct:  a custom data type (groups fields)
6          string description;                 ← what this proposal is about
7          uint256 voteCount;                   ← how many votes it has received
8      }
9
10     Proposal[] public proposals;             ← array:  dynamic list of proposals
11     mapping(address => bool) public hasVoted;   ← tracks who has already voted
12     address public owner;                    ← who controls the contract
13 }
```

New concepts:

- `struct` — lets you define a custom data type that groups related fields together. Here, each `Proposal` bundles a description and a vote count.
- `Proposal[]` — a *dynamic array* (list) of Proposal structs. You can add elements with `.push()`.
- `mapping(address => bool)` — returns `true` if an address has voted, `false` (default) otherwise.

Struc

let you model real-world objects — proposals, users, orders — as grouped data

Step 2: Access Control with Owner

```
1   address public owner;
2
3   modifier onlyOwner() {                  ← modifier: reusable access check
4       require(msg.sender == owner,        ← only the owner can pass this check
5               "Not the owner");
6       _;                                  ← _; placeholder: ‘‘run the function body here’’
7   }
8
9   constructor() {
10      owner = msg.sender;                 ← deployer becomes the owner
11  }
```

New keyword — `modifier`:

- A `modifier` wraps around a function to add a precondition check
- The `_;` (underscore semicolon) is a placeholder that means “insert the function body here”
- If the `require` fails, the function never executes

Usage: Any function marked `onlyOwner` will automatically check that the caller is the owner:

```
function doSomething() public onlyOwner { ... }
```

Constructor: Here the constructor takes no arguments — it simply records the deployer as the owner.

The owner pattern is the simplest access control — production contracts use role-based systems

Step 3: Creating Proposals

```
1    function createProposal(string memory _desc)   ← memory: temporary data location
2        public onlyOwner                            ← only the owner can call this
3    {
4        proposals.push(                              ← add a new Proposal to the array
5            Proposal(_desc, 0)                       ← create struct: description + 0 votes
6        );
7    }
```

New concept — `memory` vs `storage`:

	storage	**memory**
Where?	On the blockchain (permanent)	In RAM during function execution
Lifetime	Persists between calls	Destroyed when function ends
Cost	Expensive (storage write gas)	Cheap (temporary)
Used for	State variables	Function parameters, local vars

Why `memory` here? The `_desc` string is only needed temporarily — it gets copied into storage when `proposals.push()` executes. Solidity requires you to declare `memory` or `storage` for complex types (strings, arrays, structs) in function parameters.

Rule of thumb: function inputs use `memory`, state variables use `storage` automatically

Step 4: Casting Votes

```
1   event Voted(address indexed voter, uint256 proposalId);
2
3   function vote(uint256 _proposalId) public {
4       require(!hasVoted[msg.sender],              ← check: has this address voted before?
5               "Already voted");
6       require(_proposalId < proposals.length,     ← check: valid proposal ID
7               "Invalid proposal");
8
9       hasVoted[msg.sender] = true;                ← mark as voted (effect)
10      proposals[_proposalId].voteCount += 1;      ← increment vote count
11
12      emit Voted(msg.sender, _proposalId);        ← log the vote
13  }
```

Pattern review — Checks-Effects-Interactions:

1. **Checks:** Two `require` statements validate input
2. **Effects:** Update `hasVoted` and `voteCount`
3. **Interactions:** Emit the event (no external calls)

Why `!hasVoted[msg.sender]`? The `!` operator means "not." Since `hasVoted` defaults to `false`, the first vote passes; any subsequent call from the same address reverts.

One-address-one-vote is simple but not sybil-resistant — an attacker can create many addresses

Step 5: Reading Results

```
    function getWinner() public view
        returns (string memory winnerName, uint256 winnerVotes)
    {
        uint256 winningCount = 0;
        uint256 winningIndex = 0;

        for (uint256 i = 0; i < proposals.length; i++) {  ← loop through all proposals
            if (proposals[i].voteCount > winningCount) {   ← compare:  is this the highest?
                winningCount = proposals[i].voteCount;
                winningIndex = i;
            }
        }

        winnerName = proposals[winningIndex].description;  ← return winner's description
        winnerVotes = winningCount;                        ← return winner's vote count
    }
```

New concepts:

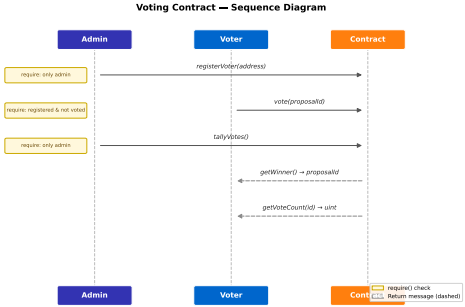
- `for` loop — iterates from `i = 0` up to `proposals.length`
- `if` statement — compares the current proposal's vote count against the running maximum
- Named return values — `winnerName` and `winnerVotes` are automatically returned

Gas note: This function is `view`, so it is free to call. But if the proposals array were very large, the loop would use more gas in a transaction context.

Loops in Solidity can be dangerous — unbounded loops may exceed the block gas limit

The Complete Voting Contract

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

contract SimpleVoting {
    struct Proposal {
        string description;
        uint256 voteCount;
    }

    Proposal[] public proposals;
    mapping(address => bool) public hasVoted;
    address public owner;

    event Voted(address indexed voter,
                uint256 proposalId);

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

    constructor() {
        owner = msg.sender;
    }

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

    function vote(uint256 _proposalId) public {
        require(!hasVoted[msg.sender], "Already voted");
        require(_proposalId < proposals.length,
```



What this contract CAN do:

- ✓ Owner creates proposals
- ✓ One vote per address
- ✓ Transparent results
- ✓ Event logging

What it CANNOT do:

- × Delegated voting

Deploying with Remix

Visual Break

cartoon/cartoon.pdf

Deploying to Testnet + Interaction

From Remix VM to a real testnet:

1. **Install MetaMask:** Browser extension wallet (`https://metamask.io`)
2. **Add Sepolia testnet:** Settings → Networks → Add Sepolia (or it may appear automatically)
3. **Get test ETH:** Visit a faucet (e.g., `https://sepoliafaucet.com`) — free test ETH sent to your wallet (requires a small `msg.value` deposit on some faucets)
4. **Connect Remix:** In "Deploy & Run," change Environment from "Remix VM" to "Injected Provider – MetaMask"
5. **Deploy:** MetaMask pops up asking you to confirm the transaction — click "Confirm" and wait for mining

Interacting with your deployed contract:

- **Blue buttons** = read functions (`view`/`pure`) — free, instant
- **Orange buttons** = write functions — require gas, need MetaMask confirmation
- **Event logs:** Visible in the Remix console and on block explorers like Etherscan

Block explorer: After deploying, paste your contract address at `https://sepolia.etherscan.io` to see all transactions and events.

Testn

use fake ETH — deploy as many contracts as you want for free

Key Takeaways + Questions

What you learned today:

1. **Solidity fundamentals:** pragma, contract, state variables, functions, view/pure, data types
2. **ERC-20 from scratch:** mapping, constructor, transfer, events, approve/transferFrom
3. **Voting with access control:** struct, modifier, memory/storage, loops
4. **Remix deployment:** compile, deploy to VM and testnet, interact with contracts

Questions for reflection:

1. Why does the ERC-20 standard require both `transfer` and `transferFrom`?
2. What happens if you forget the `require` check in the `vote` function?
3. How would you add a time limit to the voting contract?
4. Why is the Checks-Effects-Interactions pattern important for security?

Next lesson: L11 — Building DeFi (DEX, escrow, and security patterns)