Solidity Basics

Lesson 10: Summary

Prof. Joerg Osterrieder

Spring 2026

L10 Summary: Solidity Basics

Four sections covered today:

1. **What is Solidity?** — Language, compiler pipeline, EVM, and gas
2. **Your First Contract** — pragma, contract, state variables, functions
3. **ERC-20 Token** — mapping, constructor, transfer, events, approve/allowance
4. **Voting Contract** — struct, modifier, memory/storage, loops
5. **Deploying with Remix** — compile, deploy, testnet, interact

Key shift from previous lessons:

- L07 asked *what* smart contracts do
- L10 asks *how* to write them in Solidity

No prior programming experience assumed — every keyword was explained on first use

Section 1: What is Solidity?

Solidity — A statically typed language created by Gavin Wood in 2014, designed specifically for Ethereum smart contracts.

Compilation pipeline:

Key concepts:

- **EVM** (Ethereum Virtual Machine) — the decentralised computer that runs bytecode on every Ethereum node
- **Gas** — every operation costs gas (paid in ETH); prevents spam and infinite loops
- Storage writes ≈20,000 gas; arithmetic ≈3–5 gas; reads from outside = free
- Statically typed (every variable must declare a type); no floating-point numbers

---

Solidity is used in over 80% of Ethereum contracts — the dominant smart contract language

Five keywords every Solidity file uses:

Keyword	Meaning
`pragma solidity ^0.8.20`	Declare compiler version
`contract Name { }`	Define a smart contract (like a class)
`uint256 public x`	State variable — stored permanently on the blockchain
`function f() public view returns (T)`	Read-only function: `view` = reads state, free to call
`function f() public pure returns (T)`	Computation-only: `pure` = reads nothing, free to call

Pattern: state variable + write function + read function = foundation of every contract

SimpleStorage in one line: stores a number on-chain; `set(n)` costs gas, `get()` is free.

The ^ in `^0.8.20` means "compatible with" — accepts 0.8.20 and above, but not 0.9.0

ERC-20 interface — 6 required functions + 2 events:

Function	Purpose
`totalSupply()`	Tokens in existence
`balanceOf(addr)`	Balance of an address
`transfer(to, amt)`	Send tokens
`approve(spender, amt)`	Authorise a spender
`allowance(owner, sp)`	Check approved amount
`transferFrom(from,to,amt)`	Spend on behalf

Key Solidity concepts introduced:

- `mapping(address => uint256)` — key–value store; $O(1)$ lookup
- `constructor` — runs once at deploy; sets initial supply
- `msg.sender` — address of the current caller
- `require(cond, "msg")` — revert if condition fails
- `event` + `emit` — cheap log entries for wallets
- Nested mapping for allowances: `mapping(address => mapping(address => uint256))`

The approve+transferFrom pattern is how every DEX and DeFi protocol moves tokens on your behalf

Section 4: Voting Contract

SimpleVoting introduces four new Solidity patterns:

struct — custom grouped data type:

```
struct Proposal {
    string description;
    uint256 voteCount;
}
```

modifier — reusable access check:

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

memory vs. storage:

- `storage` — permanent, on-chain, expensive
- `memory` — temporary, during function call, cheap
- Function parameters for complex types require explicit `memory`

for loop + array:

- `Proposal[] proposals` — dynamic array
- `.push()` to add elements
- Loop to find winner — $O(n)$, free since `view`

Access control: deployer = owner; only owner creates proposals

---

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

Key Formulas and Patterns

ERC-20 Interface Checklist:

- ✓ `totalSupply()` — view
- ✓ `balanceOf(address)` — view
- ✓ `transfer(to, amount)` — returns bool
- ✓ `approve(spender, amount)` — returns bool
- ✓ `allowance(owner, spender)` — view
- ✓ `transferFrom(from, to, amount)` — returns bool
- ✓ `Transfer(from, to, value)` — event
- ✓ `Approval(owner, spender, value)` — event

Checks-Effects-Interactions (CEI) Pattern:

1. **Checks** — validate all inputs with `require`
2. **Effects** — update contract state (balances, flags)
3. **Interactions** — call external contracts last

Why CEI matters: updating state *before* external calls prevents reentrancy attacks.

Visibility summary:

`public`	anyone can call
`external`	only outside callers
`internal`	this contract + children
`private`	this contract only

CEI is the most important security pattern in Solidity — always check before you change state

Section 5: Deploying with Remix

Remix IDE (`https://remix.ethereum.org`) — free, browser-based, no installation required.

Deployment workflow (5 steps):

1. **Write** — create a `.sol` file in Remix's file explorer
2. **Compile** — Solidity compiler tab → select version → "Compile"
3. **Deploy** — "Deploy & Run" tab → Remix VM (local testnet) → "Deploy"
4. **Interact** — blue buttons = free reads; orange buttons = paid writes
5. **Testnet** — connect MetaMask → Sepolia testnet → get free ETH from faucet

Reading results in Remix:

- **Blue buttons** = `view`/`pure` — no gas, instant
- **Orange buttons** = state-changing — MetaMask confirmation required
- Block explorer: `https://sepolia.etherscan.io` for deployed contract history

Remix VM provides multiple test accounts with 100 ETH each — experiment without real funds

Next Lesson: L11 — Building DeFi

L10 gave you the tools. L11 puts them to work in real DeFi protocols.

From tokens to protocols — what L11 covers:

- **Decentralised Exchange (DEX)** — automated market maker using the constant-product formula $x \cdot y = k$
- **Lending Protocol** — collateralised borrowing, liquidation logic
- **Escrow** — trustless payment with conditional release
- **Security Patterns** — reentrancy guards, access control, upgrade patterns
- **OpenZeppelin Libraries** — using audited, production-grade code

Prerequisites review — make sure you can:

1. Write a complete ERC-20 token from scratch
2. Use `modifier`, `struct`, `mapping`, `event`
3. Deploy and interact with contracts on Remix
4. Explain the Checks-Effects-Interactions pattern

L11: Building DeFi — from tokens to protocols; prerequisite: this lesson

Thank You

Questions?