OST
Eastern Switzerland
University of Applied Sciences

Blockchain (BlCh)

Solidity

Thomas Bocek

02.10.2025

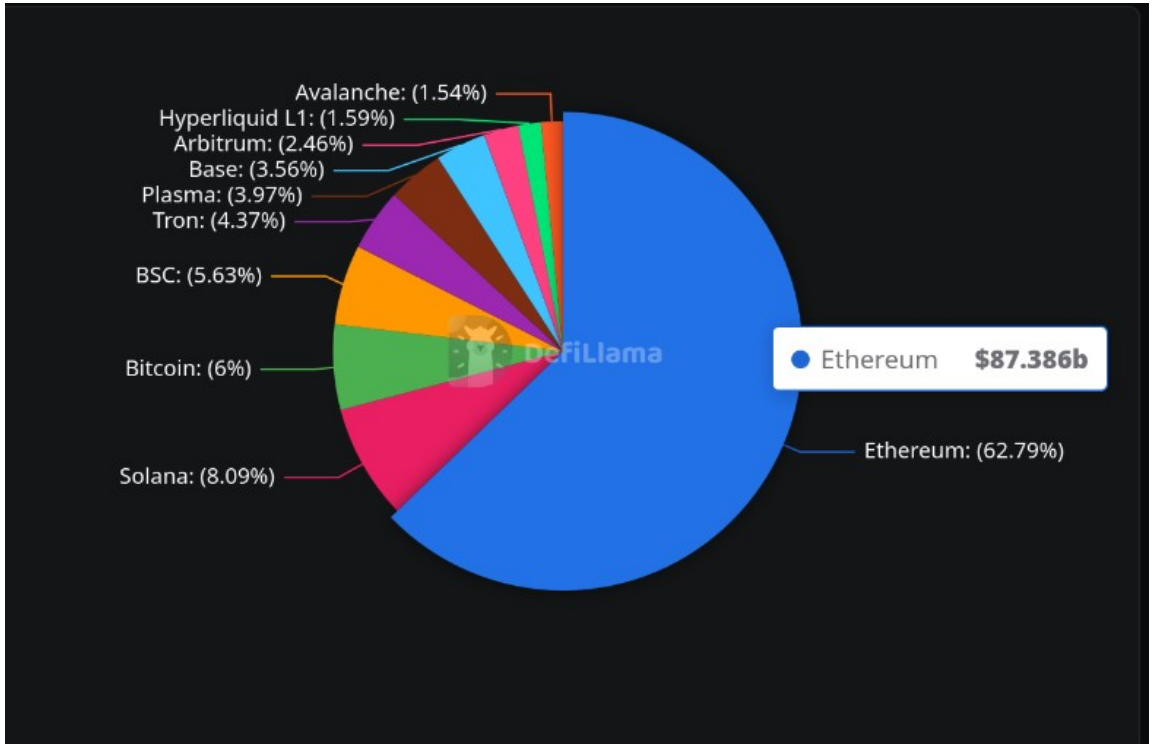
Learning Goals

- Lecture 4
  - Solidity fundamentals
    - Understand smart contract development
    - Learn Solidity syntax and structure
    - Grasp gas optimization principles
    - Deploy contracts

OST

Why Solidity? EVM Dominance

- Ethereum: ~60% of total DeFi TVL (DefiLlama, 2025)
- Top 10 chains: 7 are EVM-compatible
  - 1. Ethereum (EVM), 4. BSC (EVM), 6. Plasma (EVM), 7. Base (EVM L2), 8. Arbitrum (EVM L2), 10. Avalanche (EVM)
- Major EVM L2s: Base, Arbitrum, Optimism, Polygon
- Ethereum + EVM L2s = dominant smart contract platform
  - Most DeFi protocols built on EVM
- Learning Solidity = access to largest blockchain ecosystem



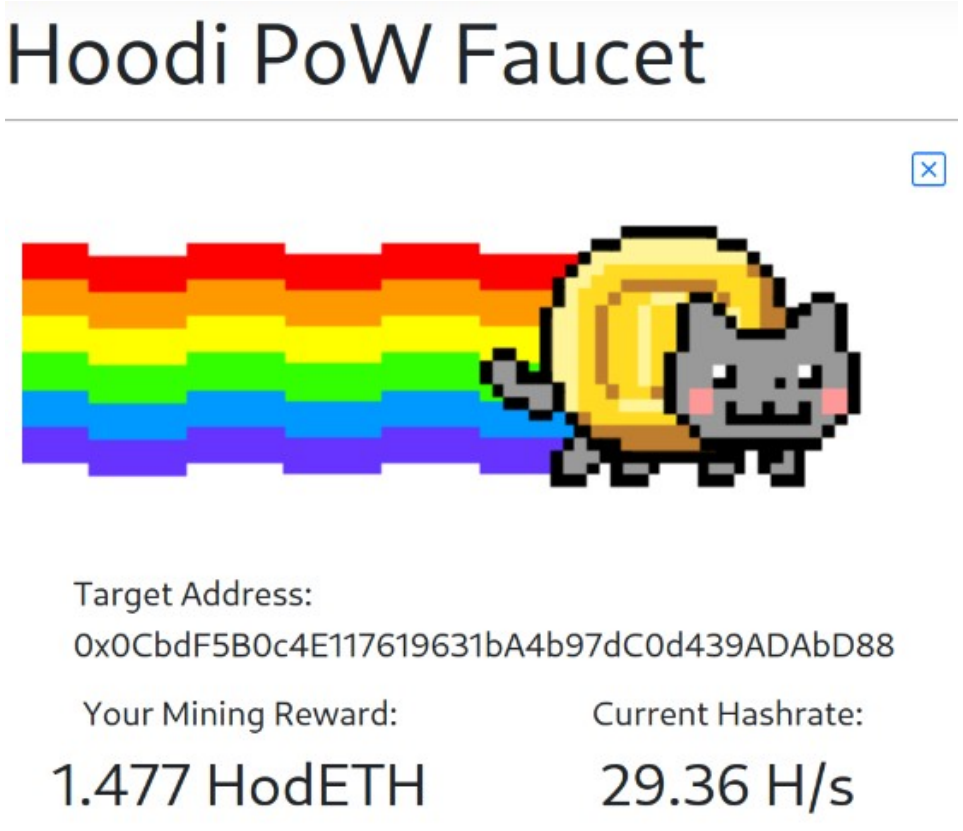
https://defillama.com/chains

OST

Development Environment

- Remix IDE: https://remix.ethereum.org
  - Browser-based, zero installation required
  - Built-in compiler, debugger, and deployment tools
- Local alternatives: VS Code, Zed (plugin), Hardhat, Foundry
- Start on testnets
  - Unless chain is extremely cheap (e.g., Sui)
- Multiple testnets exist for different purposes
  - Sepolia: Most stable, widely supported (general development)
  - Holesky being deprecated September 2025 due to issues (designed for staking, infrastructure, and protocol testing)
  - Hoodi: New testnet for staking/validators (replaces Holesky, designed for validators and staking providers)

- Free test ETH from faucets
  - **Sepolia**: https://sepolia-faucet.pk910.de/
  - Hoodi: https://hoodi-faucet.pk910.de



OST

Gas - Ethereum's Fuel

- Remix Desktop version [link]
  - No direct MetaMask integration → WalletConnect

- Payment for computation in EVM
- Every operation (opcode) costs gas
- Unit of computational work measurement
- If gas runs out: state reverted, ETH lost
- Writing to state is expensive

- Gas cost examples (yellow paper):
  - SSTORE (storage write): 20’000 gas
  - SLOAD (storage read): 200 gas
  - ADD operation: 3 gas
  - Contract creation: 32’000 gas base
- Poor optimization = high user costs = bad UX
  - 1st priority: correctness of the contract
  - 2nd priority: gas efficient

OST

Solidity Source File Structure

- SPDX License Identifier (mandatory since 0.6.8)
  - `// SPDX-License-Identifier: MIT`
  - Private code: UNLICENSED
- Version pragma:
  - `pragma solidity ^0.8.30;`
- ^0.8.30 means: ≥0.8.30 but <0.9.0
- Imports: import "filename";
  - OpenZeppelin: battle-tested, audited standard library
  - import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
  - Do not rewrite standards from scratch - use OpenZeppelin
- Comments: // single-line or /* */ multi-line

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.30;

import {ERC20} from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import {Ownable} from "@openzeppelin/contracts/access/Ownable.sol";

//Create a token
/*contract OSTToken is ERC20, Ownable {
    constructor() ERC20("OSTToken", "OST") Ownable(msg.sender) {
        _mint(msg.sender, 1000 * 10 ** decimals());
    }
}*/
```


OST

Contract Structure - State Variables / Functions

- State Variables:
  - Persistent storage on blockchain
  - Expensive to write, cheap to read

```
contract SimpleStorage {
    uint256 storedData;
}
```

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.30;
3
4  contract SimpleStorage {
5      uint256 private storedData;
6      function getStoredData() public view returns (uint256) {   2431 gas
7          return storedData;
8      }
9      function setStoredData(uint256 _storedData) public {   22492 gas
10         storedData = _storedData;
11     }
12 }
```

- Functions
  - Internal/external calls
  - Visibility: public, private, internal, external
  - Function types:
    - pure: no state read/write
    - view: read state, no write
    - payable: can send/receive ETH
    - (default): can read and write state

OST

Data Types

- Value Types
  - `bool`: true/false
  - `uint8` to `uint256` (8-bit steps), int8 to `int256`
    - `uint256` is most common (alias: `uint`)
  - `address`: 20-byte Ethereum address
  - `address payable`: can receive ETH via `.transfer()` or `.send()`
- Reference Types
  - `bytes1` to `bytes32`: fixed-size byte arrays
  - `bytes`: dynamic byte array
  - Arrays: `uint[]` dynamic or `uint[5]` fixed
  - `string`: UTF-8 encoded text (expensive!)
- Complex Types
  - Mapping: `mapping(address => uint)`
    - Hash table, no iteration possible
    - cannot get list of keys
    - Default value for non-existent keys
  - Struct: `struct User { string name; uint age; address wallet; }`
    - Custom composite types for grouped data
  - Enum: `enum Status { Pending, Active, Completed, Cancelled }`
    - Named constants for state management

OST

Function Modifiers / Events

- Execute checks before function
- Reusable access control
- Common in OpenZeppelin contracts (Ownable, Pausable)
- _; placeholder = where function body executes

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

- Events
  - One-way communication to outside world (blockchain → frontend)
  - Used for logging and frontend notifications
  - Not for reading data back into contract
  - Misused for debugging → use Hardhat console.log when local, remote - Tenderly, or Remix Debugger
  - Events are cheaper than storage (logged, not stored in state)
    - `event HighestBidIncreased(string msg);`
    - `emit HighestBidIncreased("hello");`

OST

Error Handling / Data Location

- Custom errors (gas efficient)
  - `error NotEnoughFunds(uint256 requested, uint256 available);`
  - `revert NotEnoughFunds(amount, balance);`
- require(): more expensive, common
  - `require(balance >= amount, "Not enough");`
- assert(): for catching bugs
- try/catch supported for external calls

- Data Location
  - `storage`: persistent on blockchain, expensive writes
    - State variables are always in storage
  - `memory`: temporary, cleared after function execution
    - Function parameters, local variables
  - `calldata`: read-only, cheapest for external function parameters
  - Critical for gas optimization and understanding copies vs references

OST

Built-in Variables / Inheritance

- Built-in Variables
  - `msg.sender`: caller address
  - `msg.value`: ETH sent with call
  - `block.timestamp`: current block time
  - `block.number`: current block
  - Units: wei, gwei, ether / seconds, minutes, hours, days
- Inheritance
  - Multiple inheritance supported
  - `virtual`: function can be overridden
  - `override`: function overrides parent
  - Always use OpenZeppelin base contracts (audited, battle-tested)
    - `contract MyToken is ERC20, Ownable, Pausable`

OST

Advanced Topics & Best Practices

- EVM Fundamentals:
  - All data types padded to 256 bits (32 bytes) in EVM
  - Function selector: first 4 bytes of keccak256(function signature)
    - Example: `transfer(address,uint256)` → `0xa9059cbb`
  - ABI encoding: how Solidity packs function calls and data
  - Understanding this helps optimize gas and debug issues
- Overflow & Gas Optimization:
  - Solidity ≥0.8.0: automatic overflow/underflow checks (safe by default)
  - Before 0.8: required SafeMath library to prevent wrap-around bugs
  - unchecked{}: disable overflow checks for gas optimization
    - Use only when mathematically certain no overflow possible
    - Saves ~100-200 gas per operation
    - Wrap around = security vulnerability, not acceptable
- Advanced Features (use sparingly):
  - Inline assembly (Yul): low-level EVM access, rarely needed
    - Use only for extreme optimization or EVM-specific operations
    - Makes code harder to audit and maintain

OST