L15: Solidity Fundamentals

Module B: Ethereum & Smart Contracts

Blockchain & Cryptocurrency Course

December 2025

Learning Objectives

By the end of this lesson, you will be able to:

- Understand Solidity's role as a smart contract language
- Declare and use fundamental data types (uint, address, bool, string, bytes)
- Write functions with appropriate visibility and state mutability modifiers
- Implement events for logging and monitoring
- Use mappings and arrays for data storage
- Apply inheritance and interfaces
- Write simple smart contracts (HelloWorld, Counter, SimpleStorage)

What is Solidity?

Solidity is a statically-typed, contract-oriented programming language:

- Created specifically for Ethereum smart contracts
- Syntax similar to JavaScript/C++
- Compiles to EVM bytecode
- Current stable version: 0.8.x (as of 2025)
- Developed by Ethereum Foundation

Key Characteristics:

- **Statically typed:** Types checked at compile time
- **Contract-oriented:** Code organized into contracts (like classes)
- **Inheritance support:** Multiple inheritance with C3 linearization
- **Libraries:** Reusable code without state
- **User-defined types:** Structs and enums

Basic Contract Structure

Every Solidity file starts with a version pragma:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract HelloWorld {
    // State variables
    string public message;

    // Constructor
    constructor(string memory initialMessage) {
        message = initialMessage;
    }

    // Function
    function setMessage(string memory newMessage) public {
        message = newMessage;
    }

    // View function (read-only)
    function getMessage() public view returns (string memory) {
        return message;
    }
}
```

Key Elements: License identifier, pragma, contract declaration, state variables, constructor, functions

SPDX License Identifier

Required at top of every Solidity file:

```
// SPDX-License-Identifier: MIT
```

Common Licenses:

- **MIT:** Permissive open-source license
- **GPL-3.0:** Copyleft license (derivatives must be open-source)
- **Apache-2.0:** Permissive with patent grant
- **UNLICENSED:** Proprietary code

Version Pragma:

- `pragma solidity ^0.8.0;` - Compatible with 0.8.0 to 0.8.x
- `pragma solidity >=0.8.0 <0.9.0;` - Range specification
- `pragma solidity 0.8.20;` - Exact version
- Prevents compilation with incompatible compiler versions

Value Types: Integers and Booleans

Integers:

- `uint` (unsigned): 0 to $2^{256} - 1$ (alias for `uint256`)
- `uint8`, `uint16`, ..., `uint256`: Sized variants (increments of 8)
- `int` (signed): -2^{255} to $2^{255} - 1$ (alias for `int256`)
- `int8`, `int16`, ..., `int256`: Sized variants

Booleans:

- `bool`: `true` or `false`
- Operators: `!` (not), `&&` (and), `||` (or), `==`, `!=`

```
contract Types {
    uint256 public largeNumber = 1000000000000000000;  // 1e18
    uint8 public smallNumber = 255;  // Max value for uint8
    int256 public signedNumber = -42;
    bool public isActive = true;
}
```

Value Types: Address

Address type holds 20-byte Ethereum addresses:

- `address`: Basic address type
- `address payable`: Can receive Ether via `transfer()` or `send()`

Address Members:

- `<address>.balance`: Returns Ether balance (in Wei)
- `<address payable>.transfer(uint amount)`: Send Ether, reverts on failure
- `<address payable>.send(uint amount)`: Send Ether, returns bool
- `<address>.call{value: amount}("")`: Low-level call with Ether

```
contract AddressExample {
    address public owner;
    address payable public recipient;

    constructor() {
        owner = msg.sender;  // Address of contract deployer
        recipient = payable(msg.sender);  // Convert to payable
    }

    function checkBalance() public view returns (uint) {
        return owner.balance;  // Balance in Wei
    }
}
```

Value Types: Bytes and Strings

Fixed-Size Byte Arrays:

- `bytes1, bytes2, ..., bytes32`: Fixed-size arrays
- More gas-efficient than dynamic bytes
- Commonly used for hashes: `bytes32 public dataHash;`

Dynamic Types:

- `bytes`: Dynamic byte array
- `string`: Dynamic UTF-8 string (no length or index access)
- More expensive in gas than fixed-size

```
contract BytesStrings {
    bytes32 public hash;  // 32 bytes, e.g., for Keccak-256 hash
    bytes public dynamicData;
    string public name = "Alice";

    function setHash(bytes32 _hash) public {
        hash = _hash;
    }

    function concatenate(string memory a, string memory b)
        public pure returns (string memory) {
        return string(abi.encodePacked(a, b));
    }
}
```

Reference Types: Arrays

Fixed-Size Arrays:

```
uint[5] public fixedArray;  // Array of 5 uints
```

Dynamic Arrays:

```
uint[] public dynamicArray;
string[] public names;

function addElement(uint value) public {
    dynamicArray.push(value);  // Append to array
}

function getLength() public view returns (uint) {
    return dynamicArray.length;
}

function removeLastElement() public {
    dynamicArray.pop();  // Remove last element
}
```

Memory vs Storage Arrays:

- `storage`: Persistent, expensive (state variable)
- `memory`: Temporary, cheaper (function scope)
- `calldata`: Read-only, cheapest (external function parameters)

Reference Types: Mappings

Key-value storage (like hash tables):

```
contract MappingExample {
    // Mapping from address to balance
    mapping(address => uint256) public balances;

    // Nested mapping (address to address to allowance)
    mapping(address => mapping(address => uint256)) public allowances;

    function updateBalance(address account, uint256 amount) public {
        balances[account] = amount;
    }

    function getBalance(address account) public view returns (uint256) {
        return balances[account];  // Returns 0 if key doesn't exist
    }

    function approve(address spender, uint256 amount) public {
        allowances[msg.sender][spender] = amount;
    }
}
```

Key Properties:

- All possible keys exist with default value (0 for uint, false for bool)
- Cannot iterate over mappings (no concept of keys array)
- Only allowed in storage (not memory or calldata)

Functions: Visibility Modifiers

Four visibility levels:

1. **public:** Callable from anywhere (external and internal)
   - Automatically creates getter for state variables
2. **external:** Only callable from outside contract (via transactions or other contracts)
   - More gas-efficient for large data (uses calldata)
3. **internal:** Only callable within contract or derived contracts (default for state variables)
4. **private:** Only callable within contract (not derived contracts)

```
contract Visibility {
    uint private secretNumber;
    uint internal internalNumber;

    function publicFunc() public { }           // Anyone can call
    function externalFunc() external { }       // Only external calls
    function internalFunc() internal { }       // This contract + children
    function privateFunc() private { }         // Only this contract
}
```

Functions: State Mutability

Three state mutability levels:

1. **view:** Reads state but doesn't modify it
   - No gas cost when called externally (off-chain)
   - Gas cost when called by another function (on-chain)
2. **pure:** Doesn't read or modify state
   - Can only use function parameters and local variables
   - No gas cost when called externally
3. **(none):** Can read and modify state
   - Costs gas even when called externally

```
contract Mutability {
    uint public value = 10;

    function getValue() public view returns (uint) {
        return value;  // Reads state (view)
    }

    function add(uint a, uint b) public pure returns (uint) {
        return a + b;  // No state access (pure)
    }

    function setValue(uint newValue) public {
        value = newValue;  // Modifies state (no modifier)
    }
}
```

Functions: Parameters and Returns

Data Location for Reference Types:

```
contract DataLocation {
    uint[] public storageArray;  // State variable (storage)

    // calldata: read-only, cheapest for external functions
    function processCalldata(uint[] calldata data) external pure returns (uint) {
        return data[0];  // Can read but not modify
    }

    // memory: temporary, for internal use
    function processMemory(uint[] memory data) public pure returns (uint) {
        data[0] = 999;  // Can modify, but changes don't persist
        return data[0];
    }

    // storage: persistent, modifies state
    function modifyStorage() public {
        storageArray.push(42);  // Changes persist
    }

    // Named return values
    function divide(uint a, uint b) public pure returns (uint quotient, uint remainder) {
        quotient = a / b;
        remainder = a % b;
        // Implicit return of named values
    }
}
```

Events

Events enable logging for off-chain monitoring:

```
contract EventExample {
    // Declare events
    event Transfer(address indexed from, address indexed to, uint256 amount);
    event Approval(address indexed owner, address indexed spender, uint256 amount);

    mapping(address => uint256) public balances;

    function transfer(address to, uint256 amount) public {
        require(balances[msg.sender] >= amount, "Insufficient balance");

        balances[msg.sender] -= amount;
        balances[to] += amount;

        // Emit event
        emit Transfer(msg.sender, to, amount);
    }
}
```

Indexed Parameters:

- Up to 3 parameters can be `indexed`
- Indexed parameters become searchable topics
- Enables efficient filtering (e.g., "all transfers to address X")
- Non-indexed parameters stored in log data

Error Handling: require, assert, revert

Three error handling mechanisms:

1. **require(condition, message):** Validate inputs/conditions
   - Refunds remaining gas if fails
   - Use for user input validation
2. **assert(condition):** Check invariants (should never fail)
   - Consumes all gas if fails (pre-0.8.0, now refunds)
   - Use for internal errors
3. **revert(message):** Unconditional revert
   - Refunds remaining gas
   - Use in complex conditional logic

```
function transfer(address to, uint amount) public {
    require(to != address(0), "Cannot transfer to zero address");
    require(balances[msg.sender] >= amount, "Insufficient balance");

    balances[msg.sender] -= amount;
    balances[to] += amount;

    assert(balances[msg.sender] + balances[to] == totalSupply);  // Invariant
}
```

Custom Errors (Solidity 0.8.4+)

More gas-efficient than string error messages:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.4;

contract CustomErrors {
    error InsufficientBalance(uint requested, uint available);
    error Unauthorized(address caller);

    mapping(address => uint) public balances;
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    function withdraw(uint amount) public {
        if (msg.sender != owner) {
            revert Unauthorized(msg.sender);
        }

        if (balances[msg.sender] < amount) {
            revert InsufficientBalance(amount, balances[msg.sender]);
        }

        balances[msg.sender] -= amount;
        payable(msg.sender).transfer(amount);
    }
}
```

Benefits: Lower gas cost, typed parameters, better error context

Modifiers

Reusable code for function preconditions:

```
contract ModifierExample {
    address public owner;
    bool public paused = false;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Not the owner");
        _;  // Placeholder for function body
    }

    modifier whenNotPaused() {
        require(!paused, "Contract is paused");
        _;
    }

    function pause() public onlyOwner {
        paused = true;
    }

    function unpause() public onlyOwner {
        paused = false;
    }

    function criticalFunction() public onlyOwner whenNotPaused {
        // Only owner can call, and only when not paused
        // ...
    }
}
```

Inheritance

Solidity supports multiple inheritance:

```
contract Ownable {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

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

contract Pausable is Ownable {
    bool public paused;

    function pause() public onlyOwner {
        paused = true;
    }
}

contract MyContract is Pausable {
    function doSomething() public onlyOwner {
        // Inherits owner, onlyOwner, paused, pause()
    }
}
```

Key Concepts:

- `is` keyword for inheritance
- `virtual` and `override` for function overriding

Interfaces

Define contract structure without implementation:

```
interface IERC20 {
    function totalSupply() external view returns (uint256);
    function balanceOf(address account) external view returns (uint256);
    function transfer(address to, uint256 amount) external returns (bool);

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

contract MyToken is IERC20 {
    mapping(address => uint256) private _balances;
    uint256 private _totalSupply;

    function totalSupply() public view override returns (uint256) {
        return _totalSupply;
    }

    function balanceOf(address account) public view override returns (uint256) {
        return _balances[account];
    }

    function transfer(address to, uint256 amount) public override returns (bool) {
        _balances[msg.sender] -= amount;
        _balances[to] += amount;
        emit Transfer(msg.sender, to, amount);
        return true;
    }
}
```

Example: Counter Contract

Simple contract with increment/decrement functionality:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Counter {
    uint256 private count;
    address public owner;

    event CountChanged(uint256 newCount, address changedBy);

    constructor() {
        owner = msg.sender;
        count = 0;
    }

    function increment() public {
        count += 1;
        emit CountChanged(count, msg.sender);
    }

    function decrement() public {
        require(count > 0, "Counter cannot go below zero");
        count -= 1;
        emit CountChanged(count, msg.sender);
    }

    function getCount() public view returns (uint256) {
        return count;
    }

    function reset() public {
        require(msg.sender == owner, "Only owner can reset");
        count = 0;
        emit CountChanged(count, msg.sender);
```

Key Takeaways

1. **Solidity Basics:** Statically-typed, contract-oriented language compiling to EVM bytecode
2. **Data Types:** Value types (uint, address, bool, bytes) and reference types (arrays, mappings, structs)
3. **Visibility:** public, external, internal, private determine who can call functions
4. **State Mutability:** view (read-only), pure (no state access), or default (state modification)
5. **Events:** Cheap logging mechanism with indexed parameters for efficient searching
6. **Error Handling:** require (validation), assert (invariants), revert (conditional), custom errors (gas-efficient)
7. **Modifiers:** Reusable precondition checks (e.g., onlyOwner, whenNotPaused)
8. **Inheritance:** Supports multiple inheritance with virtual/override for polymorphism

Discussion Questions

1. Why is `string` more expensive than `bytes32` for storing short text?
2. When should you use `external` vs `public` for function visibility?
3. Why can't you iterate over a mapping's keys in Solidity?
4. What are the tradeoffs between using events vs storing data in state variables for historical records?
5. How does the `indexed` keyword in events affect gas costs and queryability?

Next Lesson Preview: L16 - Lab: Contract Interaction

Coming up next (hands-on lab):

- Introduction to Remix IDE
- Deploying SimpleStorage contract
- Interacting with deployed contracts (read/write functions)
- Using MetaMask with test networks
- Deploying to Sepolia testnet
- Verifying contracts on Etherscan

Preparation:

- Install MetaMask browser extension
- Create Ethereum account and save recovery phrase
- Get Sepolia testnet ETH from faucet (sepoliafaucet.com)
- Bookmark remix.ethereum.org