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:

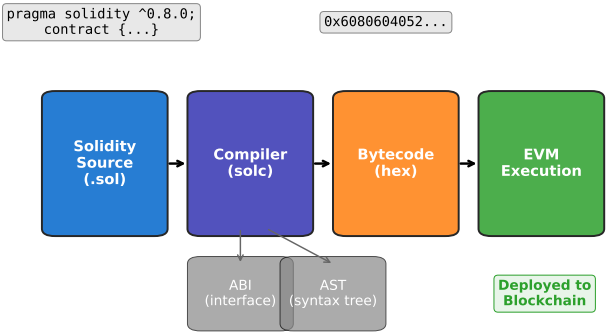
- Understand Solidity's role as a smart contract language
- Declare and use fundamental data types (uint, address, bool, string, bytes)
- Write functions with appropriate visibility and state mutability modifiers
- Implement events for logging and monitoring
- Use mappings and arrays for data storage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

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)

Solidity Compilation Pipeline



Basic Contract Structure

Every Solidity file starts with a version pragma:

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

contract HelloWorld {
    string public message;

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

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

    function getMessage() public view returns (string memory) {
        return message;
    }
}
```

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

SPDX License and Version Pragma

Required at top of every Solidity file:
Common Licenses:

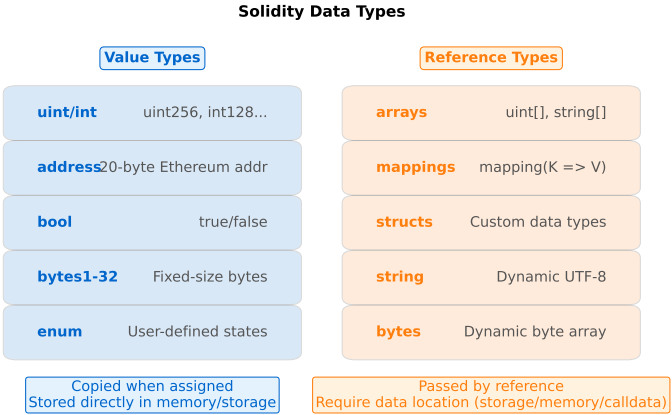
- **MIT:** Permissive open-source license
- **GPL-3.0:** Copyleft license (derivatives must be open-source)
- **UNLICENSED:** Proprietary code

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

Data Types Overview

Solidity has two categories of data types:



Value Types: Integers and Booleans

Integers:

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

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

```
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
    }
}
```

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
}
```

Reference Types: Mappings

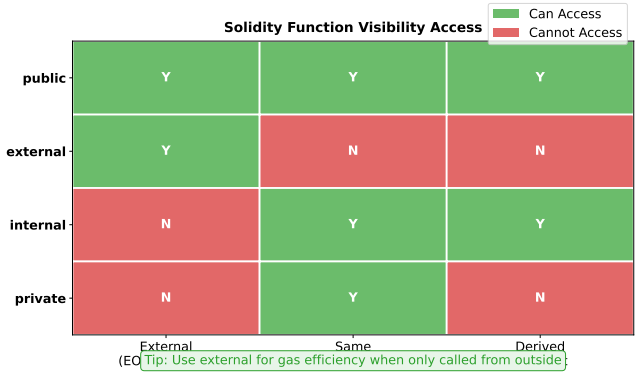
Key-value storage (like hash tables):

```
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
    }
}
```

Key Properties: All keys exist with default value, cannot iterate, only in storage

Functions: Visibility Modifiers

Four visibility levels determine who can call a function:



Functions: State Mutability

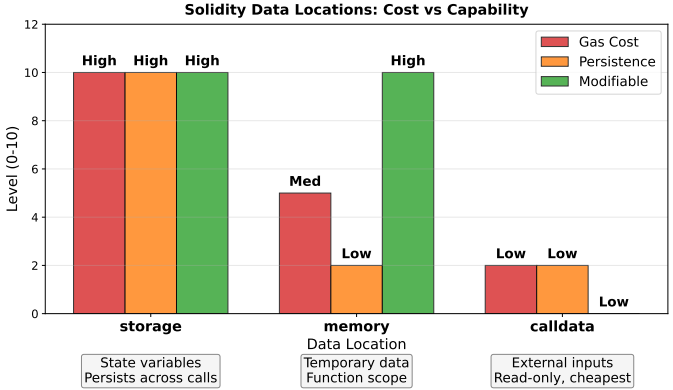
Three state mutability levels:

1. **view:** Reads state but doesn't modify it (no gas when called off-chain)
2. **pure:** Doesn't read or modify state
3. **(none):** Can read and modify state (always costs gas)

```
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)
    }
}
```

Data Locations: storage vs memory vs calldata

Reference types require explicit data location:



Events

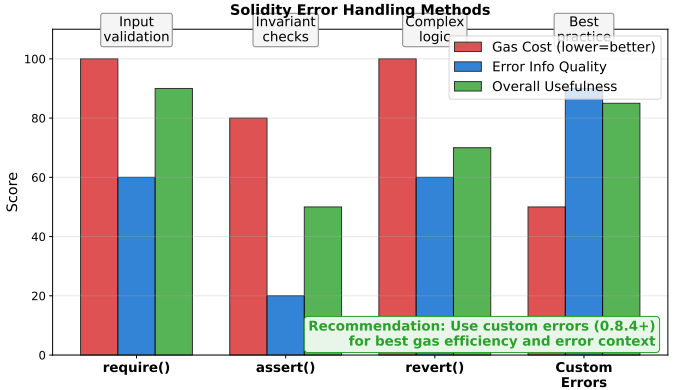
Events enable logging for off-chain monitoring:

```
contract EventExample {
    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 Transfer(msg.sender, to, amount);  // Emit event
    }
}
```

Indexed Parameters: Up to 3 parameters can be `indexed` for efficient filtering

Error Handling Methods

Solidity provides multiple error handling mechanisms:



Error Handling: require, assert, revert

1. **require(condition, message):** Validate inputs/conditions, refunds gas
2. **assert(condition):** Check invariants (should never fail)
3. **revert(message):** Unconditional revert with message

```
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:

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

    address public owner;

    function withdraw(uint amount) public {
        if (msg.sender != owner) {
            revert Unauthorized(msg.sender);
        }
        if (balances[msg.sender] < amount) {
            revert InsufficientBalance(amount, balances[msg.sender]);
        }
        // ... transfer logic
    }
}
```

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: `is` for inheritance, `virtual`/`override` for polymorphism

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;
    }
    // ... implement other functions
}
```

Key Takeaways

1. **Solidity Basics:** Statically-typed language compiling to EVM bytecode
2. **Data Types:** Value types (uint, address, bool) vs reference types (arrays, mappings)
3. **Visibility:** public, external, internal, private determine access
4. **State Mutability:** view (read-only), pure (no state), default (modify)
5. **Data Locations:** storage (persistent), memory (temp), calldata (cheapest)
6. **Error Handling:** Use custom errors (0.8.4+) for gas efficiency

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 events vs 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
- Using MetaMask with test networks
- Deploying to Sepolia testnet

Preparation:

- Install MetaMask browser extension
- Create Ethereum account and save recovery phrase
- Get Sepolia testnet ETH from faucet