L06: Solidity Programming

Extended Slides – BSc Blockchain Course

Digital Finance

2026

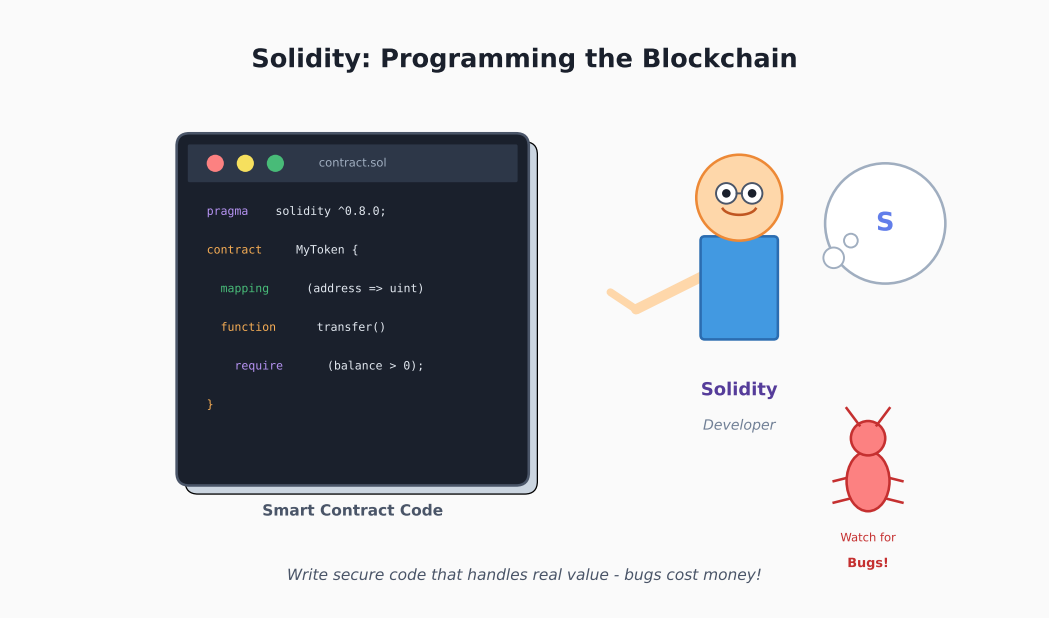
Learning Objectives

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

1. Understand Solidity contract structure and syntax
2. Implement functions with proper visibility and modifiers
3. Apply common design patterns for secure contracts
4. Identify and mitigate common security vulnerabilities
5. Deploy contracts using modern development tools

Prerequisites: L05 Ethereum and Smart Contracts.

Code That Handles Value



Purpose: Solidity is the primary language for Ethereum smart contracts. Writing secure code is critical—bugs have cost billions in lost funds.

Unlike regular software, smart contract bugs cannot be patched after deployment.

Why Solidity?

Language Characteristics:

- Statically typed, contract-oriented
- Influenced by C++, Python, JavaScript
- Compiles to EVM bytecode
- Most widely used for Ethereum

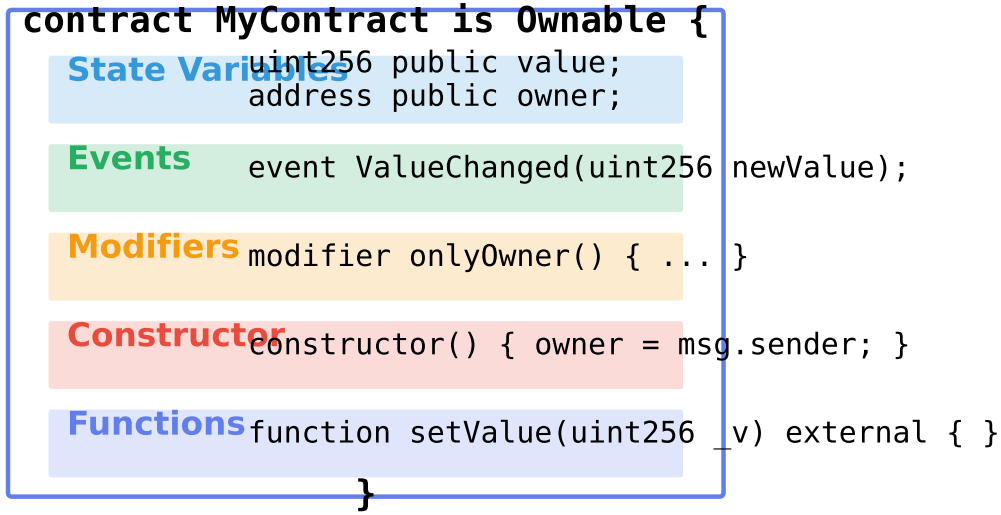
Alternatives:

- Vyper: Python-like, more restrictive
- Yul: Low-level, inline assembly
- Fe: Rust-inspired, newer

Solidity dominates the majority of deployed smart contracts.

Contract Structure

Solidity Contract Structure



Order: state vars, events, modifiers, constructor, functions.

Version Pragma

Specifying Compiler Version:

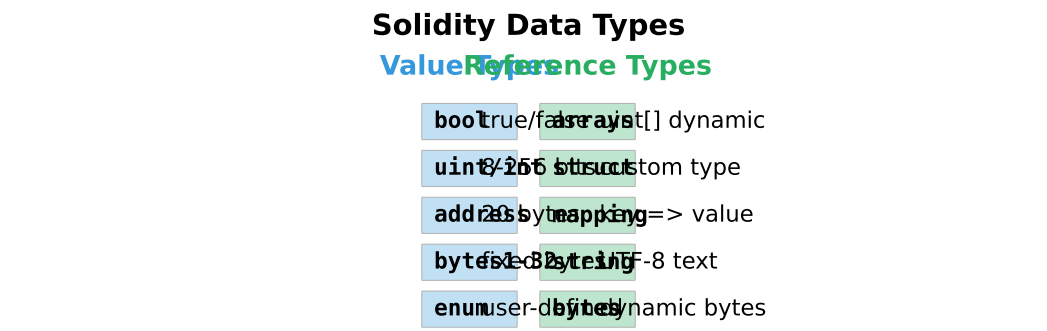
- pragma solidity ^0.8.0; (0.8.x)
- pragma solidity ¿=0.8.0 ¡0.9.0;
- pragma solidity 0.8.20; (exact)

Solidity 0.8+ Features:

- Built-in overflow checks
- Custom errors
- User-defined value types
- ABI coder v2 default

Always use 0.8+ for built-in safety checks.

Data Types



Choose smallest type that fits to save gas.

Data Locations

Storage:

- Persistent, on-chain state
- Most expensive (SSTORE: 20k–22k gas)
- Default for state variables

Memory:

- Temporary, within function
- Cleared after execution
- Default for function params

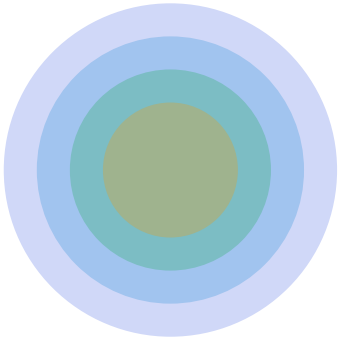
Calldata:

- Read-only, external function input
- Cheapest for reading

Use calldata for external function params when possible.

Function Visibility

Function Visibility & State Mutability



Visibility Specifiers:

`public`
Can be called internally and externally
`external`
Only external calls (saves gas)
`internal`
This contract + inheritance
`private`
Only this contract

State Mutability:
view - reads state
pure - no state access
payable - accepts ETH

External saves gas for functions only called externally.

Modifiers and Events

Modifiers

```
modifier onlyOwner() {
    require(msg.sender == owner);
    _;  // function body here
}

function withdraw() onlyOwner {
    // ...
}
```

Events

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

emit Transfer(from, to, amt);
```

Key Points:

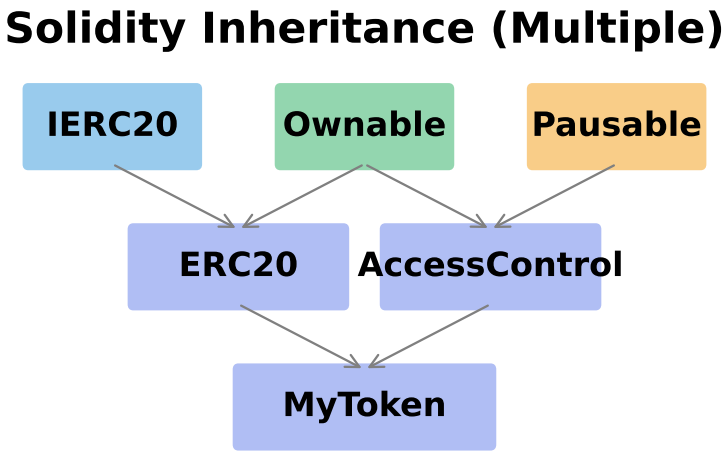
Modifiers: reusable access control, _; marks function insertion

Events: emit logs stored in tx receipt (not in state)

indexed: up to 3 params, enables efficient filtering

Events are indexed for efficient off-chain querying.

Inheritance




```
contract MyToken is ERC20, AccessControl { }
```

C3 linearization: right-most base is most derived (called first)

C3 linearization determines function resolution order.

Abstract and Interface

Abstract Contracts:

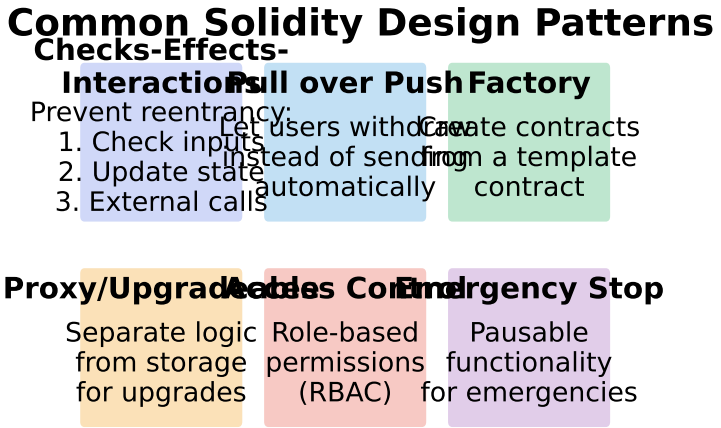
- Cannot be deployed directly
- May have unimplemented functions
- Marked with 'abstract' keyword

Interfaces:

- No state variables
- No implementations
- All functions external
- Define contract API

Interfaces enable loose coupling and composability.

Common Patterns



Learn these patterns – they prevent common bugs.

Checks-Effects-Interactions

The Pattern:

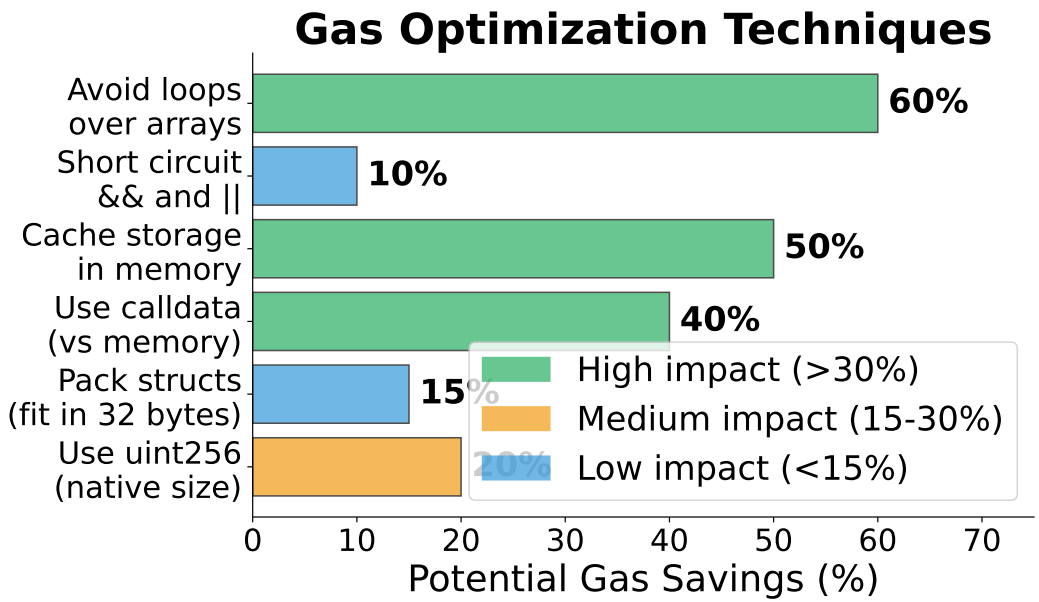
1. **Checks:** Validate inputs with require
2. **Effects:** Update contract state
3. **Interactions:** External calls last

Why It Matters:

- Prevents reentrancy attacks
- State consistent before external call
- The DAO hack exploited this

External calls can call back into your contract!

Gas Optimization



Profile with tools like Foundry's gas reports.

Storage Optimization

Packing Variables:

- uint128 + uint128 = 1 slot
- address + uint96 = 1 slot
- Order matters for packing

Caching:

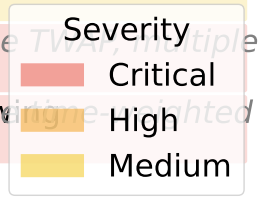
- Read storage once into memory
- Work with memory copy
- Write back at end if needed

Each SLOAD costs 2100 gas; cache repeated reads.

Security Vulnerabilities

Common Smart Contract Vulnerabilities

Vulnerability	Description	Mitigation
Reentrancy	External call before state update	*Checks-Effects-Interactions*
Integer Overflow	Unchecked arithmetic	*Use Solidity 0.8+ or SafeMath*
Access Control	Missing/weak auth checks	*Use OpenZeppelin Access*
Front-running	MEV bots see mempool	*Commit-reveal, private mempool*
Oracle Manipulation	Single price source	*Use TWAP, multiple oracles*
Flash Loan Attack	Atomic large borrow	*Volume-weighted data*



Billions lost to smart contract exploits.

Reentrancy Deep Dive

Attack Pattern:

1. Attacker calls withdraw()
2. Contract sends ETH
3. Attacker's receive() calls withdraw again
4. Repeats before balance updates

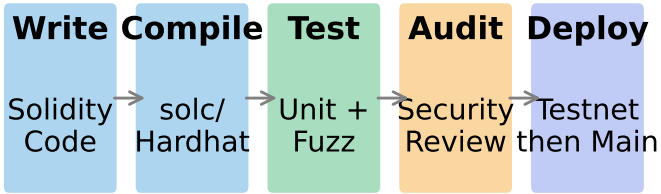
Mitigations:

- Checks-Effects-Interactions
- ReentrancyGuard (mutex)
- Pull payment pattern

The DAO lost 60M USD to reentrancy in 2016.

Deployment Flow

Smart Contract Deployment Workflow



Common Tools:

Hardhat	Foundry	Remix	Slither	Etherscan
Development	Fast testing	Browser IDE	Static analysis	Verification

Always deploy to testnet (Sepolia/Goerli) before mainnet!

Verify source code on Etherscan for transparency.

Development Tools

Frameworks:

- Hardhat: JavaScript/TypeScript
- Foundry: Rust-based, fast tests
- Brownie: Python-based

Analysis Tools:

- Slither: Static analysis
- Mythril: Symbolic execution
- Echidna: Fuzzing

Foundry gaining popularity for speed and fuzzing.

Testing Best Practices

Unit Tests:

- Test each function in isolation
- Cover edge cases
- Test reverts and events

Integration Tests:

- Test contract interactions
- Fork mainnet state
- Simulate real scenarios

Fuzz Testing:

- Random inputs
- Invariant testing

100% coverage does not mean 100% secure.

Error Handling

Solidity Error Handling

Error Handling Methods

Method	Use Case	Example	Gas Behavior
`require()`	Input validation	`require(amount > 0, "Invalid")`	Refunds remaining gas
`revert()`	Complex conditions	`if (...) revert MyError()`	Refunds remaining gas (efficient)
`assert()`	Internal invariants	`assert(balance >= 0)`	Consumes all gas (bug!)

Custom Errors (0.8.4+): More gas efficient than string messages

```
error InsufficientBalance(uint256 available, uint256 required);
```

Custom errors save 50%+ gas vs string messages.

Upgradeable Contracts

Proxy Pattern:

- Proxy holds storage + delegates calls
- Implementation holds logic
- Can swap implementation

Considerations:

- Storage layout must match
- Initialization instead of constructor
- Admin key is single point of failure

Upgradeability trades immutability for flexibility.

Key Takeaways

Remember These Points

1. Solidity: statically typed, compiles to EVM bytecode
2. Visibility levels: public, external, internal, private
3. Data locations: storage (expensive), memory, calldata
4. Checks-Effects-Interactions prevents reentrancy
5. Test thoroughly, use analysis tools, get audited
6. Custom errors more gas efficient than strings

Next Lesson: DeFi – AMMs, lending protocols, yield strategies.