# L06: Solidity Programming

## 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

*Solidity is the primary language for Ethereum smart contracts.*

**Solidity Contract Structure**

```
contract MyContract is Ownable {
    State Variables   uint256 public value;
                      address public owner;
    Events            event ValueChanged(uint256 newValue);
    Modifiers         modifier onlyOwner() { ... }
    Constructor       constructor() { owner = msg.sender; }
    Functions         function setValue(uint256 _v) external { }
}
```

*Contracts define state, events, modifiers, constructor, and functions.*

# Data Types

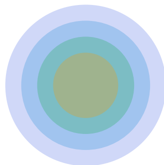
**Solidity Data Types**

| Value Types | | Reference Types | |
|---|---|---|---|
| `bool` | true/false | `arrays` | uint[] dynamic |
| `uint/int` | 8-256 bits | `struct` | custom type |
| `address` | 20 bytes | `mapping` | key => value |
| `bytes1-32` | fixed bytes | `string` | UTF-8 text |
| `enum` | user-defined | `bytes` | dynamic bytes |

***Data Locations: storage (persistent) | memory (temporary) | calldata (read-only)***

*Value types copied; reference types need explicit data location.*

# Function Visibility




*Choose minimal visibility needed for security.*

**Common Solidity Design Patterns**

**Checks-Effects-Interactions**
Prevent reentrancy:
1. Check inputs
2. Update state
3. External calls

**Pull over Push**
Let users withdraw instead of sending automatically

**Factory**
Create contracts from a template contract

**Proxy/Upgradeable**
Separate logic from storage for upgrades

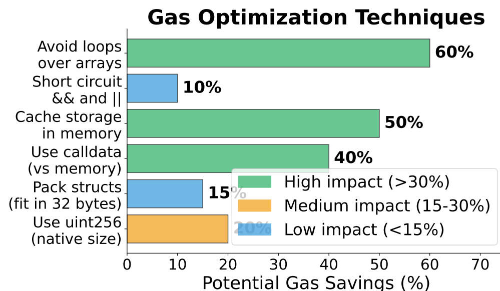
**Access Control**
Role-based permissions (RBAC)

**Emergency Stop**
Pausable functionality for emergencies

***Use OpenZeppelin implementations for battle-tested patterns***

*OpenZeppelin provides battle-tested implementations.*

# Gas Optimization




*Optimize storage access – it's the most expensive operation.*

# Security Vulnerabilities

**Common Smart Contract Vulnerabilities**

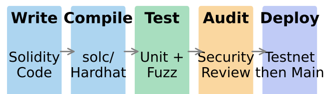
| Vulnerability | Description | Mitigation |
|---|---|---|
| Reentrancy | External call before state update | Checks-Effects-Interactions |
| Integer Overflow | Unchecked arithmetic | Use Solidity 0.8+ or SafeMath |
| Access Control | Missing/weak auth checks | Use OpenZeppelin Access |
| Front-running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

Severity
Critical
High
Medium

*Audits are essential before mainnet deployment.*

# Deployment Flow

**Smart Contract Deployment Workflow**




**Common Tools:**

| Hardhat | Foundry | Remix | Slither | Etherscan |
|---|---|---|---|---|
| Development framework | Fast testing/fuzzing | Browser IDE | Static analysis | Verification |

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

*Test extensively on testnet before mainnet.*

# 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 (!ok) revert MyError()` | Custom errors (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 gas vs string messages.*

# Key Takeaways

**Remember These Points**

1. Solidity: statically typed, contract-oriented language
2. Visibility: public, external, internal, private
3. Use Checks-Effects-Interactions pattern
4. Optimize storage access for gas savings
5. Always audit before mainnet deployment

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