#5 Solidity II: Contracts in Practice

Lecture Notes for CS190N: Blockchain Technologies and Security October 13, 2025

In this lecture, we turn Solidity from a single contract skill into a practical toolkit for production systems. We begin by structuring behavior with inheritance, using virtual and override correctly, calling into parents with super, and resolving multiple parents with deterministic linearization. We then make state changes visible with events and indexed topics, and show how clients subscribe and query without touching storage. Next, we package reusable helpers with libraries, protect critical paths with simple guards against reentrancy, and compose across protocols through clear interfaces such as IERC20. Finally, we apply essential security practices, including the Checks, Effects, Interactions sequence, pull over push value transfer, and explicit access control. By the end, you will be able to read and write small contracts that use these patterns, reason about correctness and gas, and assemble maintainable components that fit cleanly into today's Ethereum ecosystem.

1 SOLIDITY II: CONTRACTS IN PRACTICE

These notes follow the structure of the slides and aim to connect single contract patterns to production grade designs seen in DeFi. We will weave four pillars, *Inheritance*, *Events*, *Libraries with Guards*, and *Interfaces*, into a coherent path: first we reuse and specialize behavior via inheritance, then surface state changes through events, rely on libraries and lightweight guards to keep code safe and small, and finally compose with other protocols through stable interfaces. We close with security patterns that govern the order of operations in value moving code.

1.1 Motivation

Real systems are larger than any one file, and they must reuse policy across modules, make decisions visible to off chain services, and interoperate with protocols they do not control. Inheritance lets us express "is a" relationships without duplication. Events give front ends and indexers a cheap, queryable audit trail. Libraries and simple guards provide reusable safety rails. Interfaces formalize how we talk to the outside world. The overall goal is a codebase that is secure, clear in its intent, and maintainable when requirements change.

2 INHERITANCE

2.1 Why Inheritance?

Inheritance is effective when the relationship truly captures specialization. A base contract can declare state and policy once, such as ownership or pausability, so that children inherit both the data shape and the rules that protect it. This reduces duplication and clarifies who is responsible for which behavior. When the interface must remain stable but the behavior needs to evolve, overriding gives us targeted control without changing how callers interact with the contract.

2.2 Base Policy, Child Logic: `Ownable`

A minimal `Ownable` captures a simple but pervasive rule: certain functions should be callable only by an administrator. By pushing that check into a `modifier`, children avoid repeating the same `require`. The constructor fixes the initial owner at deployment, so the child's sensitive operations are protected from day one.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

contract Ownable {
    address public owner;

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

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

contract MyVault is Ownable {
    function emergencyWithdraw() external onlyOwner {
        // Admin-only logic
    }
}
```

2.3 Overriding and super

To change behavior while preserving a stable surface, mark parent functions `virtual` and child implementations `override`. This makes polymorphism explicit and catches signature drift at compile time. When the parent already encodes a safe sequence of checks and state updates, `super` lets us extend rather than replace it. This is useful for inserting logging or additional pre or post conditions without risking regressions.

Extend with `super` *(audit before core).* The following child emits an audit log before delegating to the parent's trusted core logic.

```
contract Token {
    function transfer(address to, uint a) public virtual { /* ... */ }
}

contract AuditedToken is Token {
    event TransferAudited(address f, address t, uint a);

    function transfer(address to, uint a) public override {
        emit TransferAudited(msg.sender, to, a);
        super.transfer(to, a); // call next in linearized order
    }
}
```

2.4 Multiple Inheritance and C3 Linearization

Solidity supports composing behavior from multiple parents. To keep method resolution deterministic, the language uses C3 linearization. A call to `super.foo()` moves to the next implementation in a single, well defined order, which at the current level visits rightmost parents first. When overriding a function present in several parents, the child lists all of them in the `override` clause. This makes the merge explicit to both the compiler and the reader.

```
contract Base { function foo() public virtual {} }

contract A is Base {
    function foo() public virtual override {}
}

contract B is Base {
    function foo() public virtual override {}
}

contract Final is A, B {
    function foo() public override(A, B) {
        super.foo(); // next is B.foo() (rightmost parent)
    }
}
```

3 EVENTS

3.1 Why Events?

Contracts cannot push notifications to users or indexers, and persisting long histories in storage is expensive. Events solve both issues because they are cheap, structured logs that off chain clients can subscribe to in real time or query later. A good rule of thumb is to emit after state changes so that observers see facts that already hold on chain. This aligns the mental model between storage and the logs that mirror it.

3.2 Declaring and Emitting, then Subscribing

An event declaration is the schema of what the contract promises to record. Emitting the event stores concrete values from the current execution in the transaction log. Contracts themselves do not read logs. Front ends and indexers do. That separation of concerns keeps on chain code lean while enabling rich off chain experiences.

```
contract Vault {
    event Deposit(address from, uint amount);

    function deposit() external payable {
        // ... update storage, then emit ...
        emit Deposit(msg.sender, msg.value);
    }
}
```

On the client side, a lightweight subscription is enough to react to new deposits without scanning full storage.

```
vault.on("Deposit", (from, amount) => {
    console.log(`${from} deposited ${ethers.formatEther(amount)} ETH`);
});
```

3.3 Indexed Topics for Efficient Queries

Topics act like indexes. In a non anonymous event, you can index up to three parameters because the event signature occupies the first topic. Indexing common filters such as `from` or `to` in transfers lets clients select the relevant subset directly from topics. For dynamic types, the topic stores the `keccak256` hash while the full value goes in the data section.

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

4 LIBRARIES AND GUARDS

4.1 Why Libraries?

When there is no meaningful “is a” relationship, libraries provide a better fit than inheritance. They package stateless helpers that multiple contracts can use without pulling in foreign state or behavior. Internal functions from libraries are inlined by the compiler, which keeps deployments small. Reusable guards can stay abstract but focused, so the call sites remain readable.

4.2 `SafeMath` and `ReentrancyGuard`

Even though Solidity 0.8 introduces built in overflow checks, a tiny `SafeMath` style helper is still pedagogically useful for illustrating `using for`. A minimal reentrancy guard implements a simple mutex. The modifier communicates intent at the call site and prevents overlapping entries into critical sections.

```solidity
library SafeMath {
    function add(uint a, uint b) internal pure returns (uint) {
        return a + b;
    }
}

abstract contract ReentrancyGuard {
    uint private locked; // acts as a mutex

    modifier nonReentrant() {
        require(locked == 0, "reentrancy");
        locked = 1;
        _;
        locked = 0;
    }
}
```

4.3 Using Libraries and Guards Together

Combining helpers with guards keeps arithmetic tidy and value transfers safe. The pattern below keeps the mutation close to the data and ensures the external call cannot reenter the withdrawal logic.

```solidity
using SafeMath for uint256;

contract Vault is ReentrancyGuard {
    mapping(address => uint) balances;

    function deposit() external payable {
        balances[msg.sender] = balances[msg.sender].add(msg.value);
    }

    function withdraw(uint a) external nonReentrant {
        require(balances[msg.sender] >= a, "Insufficient");
        balances[msg.sender] -= a;
        (bool ok,) = msg.sender.call{value: a}("");
        require(ok, "Transfer failed");
    }
}
```

5 INTERFACES AND COMPOSABILITY

5.1 Why Composability Matters

DeFi systems thrive because they can call each other without coordination. Interfaces formalize that contract. They describe the external functions and return types a callee promises to provide. Callers then code to the interface rather than a particular implementation, which keeps modules plug and play. This is composition, not inheritance. The two contracts are peers rather than parent and child.

5.2 Using Interfaces (`IERC20`)

An `IERC20` style interface captures the minimal surface needed for token transfers and approvals. The caller holds a reference to any contract that satisfies this ABI and interacts through it. This enables swapping implementations without changing the caller's code.

```
interface IERC20 {
    function transfer(address to, uint a) external returns (bool);
    function approve(address s, uint a) external returns (bool);
    function transferFrom(address f, address t, uint a) external returns (bool);
}

contract Treasury {
    IERC20 public token;

    constructor(address _t) {
        token = IERC20(_t);
    }

    function deposit(uint a) external {
        // caller must have approved Treasury beforehand
        token.transferFrom(msg.sender, address(this), a);
    }
}
```

6 SECURITY PATTERNS

6.1 Why Security Patterns Matter

Many real incidents trace back to ordering mistakes around external calls. Solidity gives us a few simple tools, and they work when applied consistently. The CEI discipline, which stands for Checks, Effects, Interactions, is the backbone. Validate first, update your own state next, and only then talk to the outside world. Reentrancy guards reinforce this approach by preventing overlapping entries into sensitive paths. Adopting pull over push avoids sending value to arbitrary receivers from arbitrary code paths.

6.2 CEI: Order Matters

The withdraw pattern below demonstrates the CEI sequence. Because we clear the balance before the external call, a reentrant attempt sees the updated state and fails immediately.

```
contract SecureWithdrawals {
    mapping(address => uint) public balances;

    function withdraw(uint amount) external {
        // Checks
        require(amount > 0 && balances[msg.sender] >= amount, "invalid");

        // Effects
        balances[msg.sender] -= amount;

        // Interactions
        (bool ok,) = msg.sender.call{value: amount}("");
        require(ok, "transfer failed");
    }
}
```

6.3 Guarding with a Modifier

A small modifier expresses intent succinctly. It also decouples the guard from business logic, which keeps functions clear and makes auditing easier.

```
modifier nonReentrant() {
    require(locked == 0, "locked");
    locked = 1;
    _;
    locked = 0;
}
```

6.4 Pull over Push

Rather than sending funds as a side effect of some other action, track entitlements and let users claim them. This limits the number of external calls and keeps money moving code paths explicit and short.

```
contract Rewards {
    mapping(address => uint) public pending;

    function claim() external {
        uint a = pending[msg.sender];
        pending[msg.sender] = 0; // Effects first
        (bool ok,) = msg.sender.call{value: a}(""); // Interaction last
        require(ok, "claim failed");
    }
}
```

6.5 Additional Good Practices

There are a few habits that punch above their weight. Keep access control visible at the call site, for example `onlyOwner` or role checks, and fail fast with helpful messages. Pin the compiler, for example `^0.8.x`, so that built in overflow checks stay in place. Prefer `immutable` and `constant` for configuration, and avoid hard coded addresses. Emit events for user facing actions and document any state change. Finally, test beyond happy paths. Combine unit tests with fuzzing and static analysis, and prefer forking tests when behavior depends on mainnet state.

7 KEY TAKEAWAYS

Inheritance gives us structure and policy reuse. Events make state changes visible to the world that watches. Libraries and small guards help us compose safely. Interfaces unlock composability across independent protocols. With those pillars in place, applying CEI, non reentrancy, and pull payments becomes routine rather than exceptional, which is what we want for code that moves value.

REFERENCES

A USEFUL RESOURCES

A.0.1 ConsenSys Smart Contract Best Practices. A comprehensive guide written by industry leaders, covering a wide range of security considerations, known attack patterns, and recommended development practices. https://consensysdiligence.github.io/smart-contract-best-practices/

A.0.2 Smart Contract Vulnerability Dataset. A dataset that categorizes known smart contract vulnerabilities. https://solodit.cyfrin.io/